FACULTAT D'INFORMÀTICA DE BARCELONA

Implementación de un Motor de Inteligencia Artificial para un Videojuego de Fútbol

Memoria del proyecto

Autor	Juan Ramón Hernández Pacheco
Director	Luis Solano Albajes
Presidente	Luis Pérez Vidal
Vocal	Marc Gonzàlez Tallada

Índice

Índice de Figuras

1 Introducción

1.1 Motivación

Los videojuegos se han convertido en una forma de ocio digital muy extendida. Pero a medida que aumenta el consumo de videojuegos aumenta también la exigencia del consumidor hacia el realismo de la inteligencia de los personajes que no son controlados directamente por el usuario. Para que estos personajes puedan enfrentarse o colaborar con el usuario durante el juego deben exhibir el comportamiento más apropiado a cada situación.

Dado que el desarrollo de la inteligencia artificial es una tarea compleja, es deseable un marco de trabajo que facilite su desarrollo y que permita reutilizar en lo posible los comportamientos creados tanto en el mismo videojuego como en futuros.

Por este motivo se ha decidido desarrollar un motor de inteligencia artificial que sea sencillo de integrar en un videojuego y que permita gestionar todos los personajes no controlados por el usuario así como gestionar la coordinación de los mismos.

Para poder demostrar la aplicabilidad del motor en un contexto concreto se ha optado por desarrollar un videojuego de fútbol sencillo.

El fútbol se caracteriza por ser un juego en equipo muy dinámico. Es un juego en equipo porque los futbolistas se deben de coordinar para meter más goles que el equipo contrario estableciéndose así unas relaciones de oposición y cooperación entre ellos. Es dinámico porque la posición tanto de los futbolistas como de la pelota varía continuamente.

Estas dos características determinan los dos grandes retos que se buscan resolver en el proyecto como son la coordinación entre futbolistas y la toma de decisiones en un entorno dinámico.

1.2 Objetivos

El proyecto tiene como objetivo implementar un motor de inteligencia artificial para un videojuego de fútbol capaz de gestionar la inteligencia artificial de los futbolistas de ambos equipos.

Para poder utilizar el motor, se desarrollará un videojuego donde se integrará dicho motor para poder demostrar su correcto funcionamiento. El videojuego estará formado por un motor gráfico, un motor de físicas y el juego de fútbol. Como el objetivo es la implementación del motor de inteligencia artificial, se utilizarán un motor gráfico y un motor de físicas de código libre y el resto será código propio.

El proyecto se ha dividido en las siguientes fases:

- **Desarrollo del juego e integración de los motores gráficos y de físicas.** El objetivo es implementar el videojuego donde poder integrar el motor de inteligencia artificial implementado.
- **Desarrollo del motor de inteligencia artificial.** El objetivo es implementar el motor de inteligencia artificial capaz de gestionar:
  - El conocimiento de los personajes.
  - Toma de decisiones a corto plazo.
  - Toma de decisiones a largo plazo.
  - Tácticas colectivas.
- **Desarrollo de la inteligencia artificial específica para los futbolistas**. El objetivo es implementar un conjunto suficiente de comportamientos y tácticas colectivas que permita demostrar el correcto funcionamiento del motor implementado.

2 Análisis de antecedentes

2.1 Videojuegos

Un videojuego es un juego electrónico que a partir de la interacción con una interfaz de usuario se genera una respuesta visual en el dispositivo electrónico que lo ejecuta.

Actualmente el tipo de dispositivos donde se ejecutan los videojuegos, también conocidos como plataformas, es muy variado, desde un ordenador o videoconsola a un teléfono móvil.

La interacción del usuario con el videojuego puede variar en función de la plataforma en que se ha desarrollado. En los videojuegos para ordenador es habitual interactuar con un teclado y ratón, pero también es posible interactuar con otros periféricos como pueden ser los volantes o joysticks. En cambio en un teléfono móvil la interacción es a través del teclado del móvil o de la pantalla táctil variando así la experiencia del jugador.

Los videojuegos también se pueden clasificar por géneros en función de su temática, jugabilidad, estética, etc. Una clasificación habitual es la siguiente:

- **Acción.** Los videojuegos de acción se caracterizan por la velocidad de respuesta requerida por el usuario. Prevalecen los reflejos sobre el razonamiento y la estrategia.

Figura 1: Captura de juegos de acción

- **Aventura.** En este tipo predominan los diálogos y la interacción con el resto de personajes y elementos. Suelen ser lineales e incluyen la superación de enigmas, problemas lógicos, etc.

Figura 2: Captura de juegos de aventura

- **Deportes**. Son juegos que emulan la práctica de algún deporte como el fútbol, baloncesto, golf o la conducción de vehículos.

Figura 3: Captura de juegos de deporte

- **Estrategia.** Son aquellos juegos donde prevalecen la estrategia y la táctica. Es habitual coordinar varios personajes a la vez para alcanzar los objetivos.

Figura 4: Captura de juegos de estrategia

- **Rol.** El jugador encarna a un personaje asumiendo y desarrollando su personalidad e interaccionando con el resto de elementos. También se caracteriza por el control exhaustivo de sus constantes vitales, habilidades y otras características propias del personaje.

Figura 5: Captura de juegos de rol

- **Simuladores.** Son juegos que emulan aspectos variados de la realidad con mucho tipo de detalle.

Figura 6: Captura de juegos de simulación

2.1.1 Desarrollo de videojuegos

El desarrollo de un videojuego es similar al desarrollo de un programa convencional con la diferencia de la gran cantidad de aporte creativo como la historia, la música o el diseño de personajes y entornos. Por lo tanto, el desarrollo de videojuegos es una actividad multidisciplinaria en la que participan profesionales de la informática, del diseño o del sonido entre otros.

Es habitual que el desarrollo de videojuegos siga un proceso iterativo basado en prototipos donde en cada prototipo se añaden y se prueban las nuevas

funcionalidades. De esta manera es más fácil experimentar los diferentes algoritmos y la jugabilidad del videojuego. Este proceso iterativo se puede aplicar en paralelo al desarrollo del programa y al desarrollo de los contenidos.

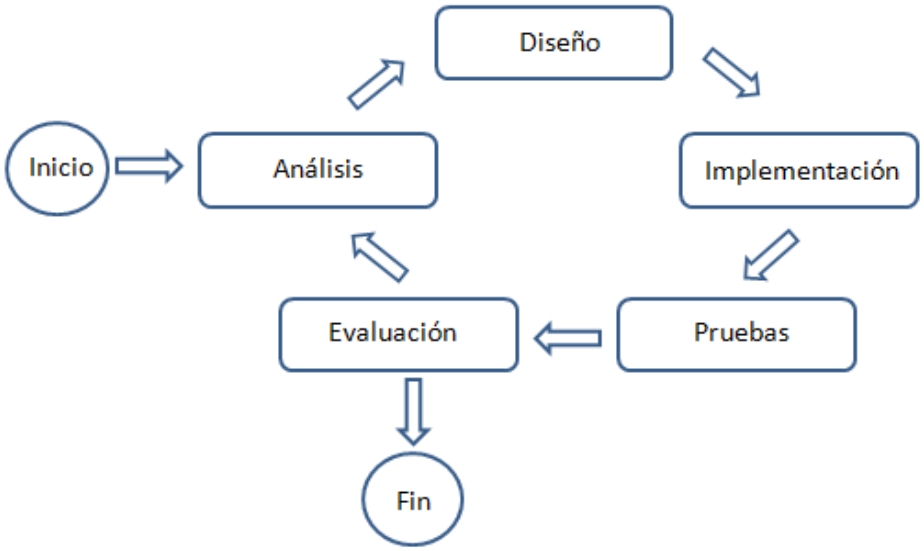


Figura 7: Diagrama de desarrollo en espiral

Desarrollo del programa

Un videojuego como programa se puede descomponer en varios subsistemas especializados que interactúan unos con otros.

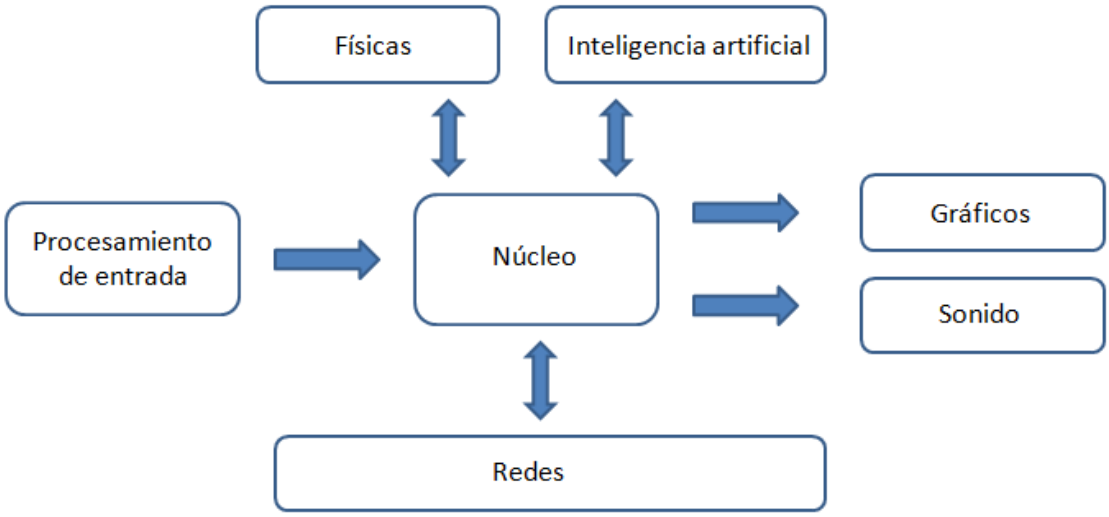


Figura 8: Diagrama de subsistemas de un videojuego

- **Núcleo del juego.** Este componente es el que mantiene el estado del videojuego, contiene las reglas del juego así como la gestión de las entidades que participan en él.
- **Gráficos**. Es el componente encargado de representar visualmente el estado del videojuego representado en el núcleo del juego.
- **Físicas.** Este componente se encarga de gestionar la dinámica de los objetos así como detectar las colisiones entre los mismos y notificarlo al núcleo del juego para modificar su estado.
- **Procesamiento de entrada**. Es el encargado de capturar los eventos producidos por los diferentes tipos de periféricos, procesarlos y notificar al núcleo del juego.
- **Inteligencia artificial.** Es el encargado de gestionar la toma de decisiones sobre qué acciones deben realizar los personajes no controlados por el jugador.
- **Redes.** Es el componente encargado de gestionar la comunicación entre varios dispositivos que ejecutan el mismo videojuego. Permite a varios jugadores colaborar o competir entre ellos.
- **Sonido.** Este componente se encarga de reproducir la música, las voces o los efectos de sonidos a través de un dispositivo de reproducción de sonido.

Desarrollo de contenidos

Paralelamente al desarrollo del videojuego como programa, se desarrollan todos sus contenidos artísticos.

- **Historia.** Muchos videojuegos requieren de una historia atractiva que atrape al jugador. Se describe como se desenvolverán los personajes del juego así como la historia del mundo.
- **Ilustración, modelado y animación.** En los videojuegos en 2D se requiere de la elaboración de las ilustraciones de los personajes mientras que en 3D

se desarrollan sus modelos 3D. En ambos casos se elaboran las animaciones así como los escenarios donde se desarrolla toda la acción.

- **Interfaz.** La gran mayoría de videojuegos disponen de una interfaz gráfica de usuario que muestra información relativa al juego y que le permite al usuario interactuar con él.
- **Sonidos.** Es necesario crear sonidos para cada objeto, personaje o efecto así como la música y sonidos de ambiente.

2.2 Inteligencia artificial

La inteligencia artificial académica se define como la creación de programas que emulan actuar y pensar como las personas, así como actuar y pensar racionalmente. Sin embargo, la inteligencia artificial en los videojuegos se centra en la creación de código que hace que los personajes controlados por el programa parezcan tomar decisiones inteligentes cuando existen múltiples alternativas dada una situación, resultando en un comportamiento relevante, efectivo y útil.

En los primeros videojuegos, la programación de la inteligencia artificial era más conocida como programación de jugabilidad, porque no había nada de inteligente en el comportamiento de los personajes controlados por el programa ya que se exhibían comportamientos basados en patrones o en movimientos repetitivos. Esto era debido a las limitaciones de la velocidad de los procesadores y de memoria. Los patrones se podían almacenar fácilmente, requerían poco código para poder ser ejecutados y no necesitaban cálculos.

Otra técnica utilizada en el pasado era hacer que los personajes parecieran inteligentes permitiéndoles hacer trampas, teniendo información adicional sobre el juego que en cambio el jugador no tenía y permitía la toma de decisiones parecer inteligente. Otras técnicas basadas en las trampas consistían en proveer a los personajes de habilidades, recursos extras, etc. Esta técnica permite tener unos oponentes competitivos pero no hacen la experiencia del juego divertida dado que le da al jugador la sensación de que sus oponentes hacen cosas que él no va a poder hacer nunca.

En la actualidad, estas técnicas están siendo abandonadas gracias al aumento de la velocidad de los procesadores y de la memoria disponible. Además de estas mejoras tecnológicas, hay un incremento del tiempo de procesador dedicado a la inteligencia artificial. Esto es posible gracias a que se han creado unidades especializadas para los gráficos liberando así al procesador.

Esto implica que los personajes controlados por el videojuego pueden encontrar mejores soluciones en entornos más complejos sin el uso de trampas. Cada vez es más habitual el uso de técnicas de la inteligencia artificial académica como son las

búsquedas heurísticas, el aprendizaje, la plantificación, etc. reduciéndose así la diferencia entre la inteligencia artificial en los videojuegos y la académica.

2.2.1 Agentes inteligentes

Se define como agente inteligente a una entidad autónoma y flexible que observa y actúa sobre un entorno con la finalidad de cumplir sus objetivos. Se entiende por flexible a la capacidad ser reactivo, pro-activo y social. Un agente es reactivo cuando es capaz de responder a los cambios de su entorno, es pro-activo cuando es capaz de intentar cumplir sus propios objetivos y es social cuando es capaz de comunicarse con otros agentes.

Encontrar el equilibro entre reactividad y pro-actividad es un problema complejo y de echo sigue siendo un problema de investigación. Cuanto más tiempo se dedica a razonar sobre los objetivos a largo plazo menos tiempo se tiene para reaccionar y al revés.

En un agente se identifican 4 elementos: el entorno, la capacidad de percepción, capacidad de acción y la toma de decisiones.

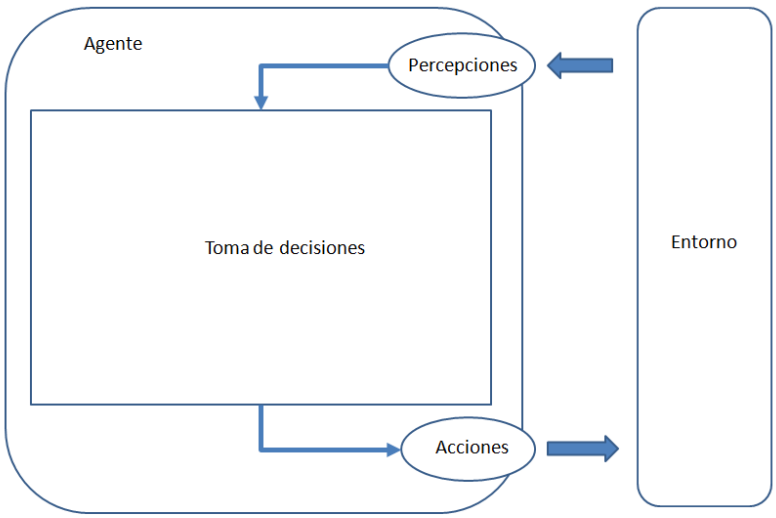


Figura 9: Arquitectura de un agente inteligente

- **El entorno.** El entorno es el espacio al que pertenece el agente y donde ocurren los todos los sucesos. Las características del entorno condicionan

en gran medida el tipo de agente que se va a construir. Podemos encontrar los siguientes tipos de entornos:

- Accesible/No accesible. Un entorno es accesible si el agente es capaz de percibir el estado completo del entorno.
- Determinista/No determinista. Un entorno es determinista si es posible conocer a partir del estado actual y la decisión tomada por el agente, el estado futuro del propio entorno y del mismo agente.
- Episódico/No episódico. Un entorno es episódico si es posible dividir el estado del agente en episodios con características propias.
- Estático/Dinámico. Un entorno es estático si solo se altera por la acción del agente.
- Discreto/Continuo. Un estado es discreto cuando es posible concretar todos los estados del entorno.

- **La capacidad de percepción.** La capacidad de percepción está definida por los elementos que es capaz de reconocer un agente. El número de elementos que es capaz de reconocer y su naturaleza determina la cantidad de información que sabe el agente sobre su entorno. Si algo no se percibe, el agente no es capaz de saber de su existencia.
- **La capacidad de acción**. La capacidad de acción está definida como el conjunto de movimientos que puede realizar un agente para modificar su entorno. La única manera que tiene el agente de modificar su entono es a través de las acciones que puede realizar.
- **Toma de decisiones.** Un agente debe ser capaz de decidir qué acción realizar en función de las percepciones que tiene sobre su entorno. La manera en como realiza esta decisión determina la arquitectura interna del agente. Podemos encontrar los siguientes tipos de agentes:
  - Agentes reactivos. Un agente reactivo actúa solo en función de sus percepciones actuales, ignorando el historial de percepciones. La toma de decisiones se basa en un conjunto de reglas condición-acción.

Este tipo de agentes solo son aplicables cuando el entorno es totalmente accesible porque conocen en todo momento el estado del entorno.

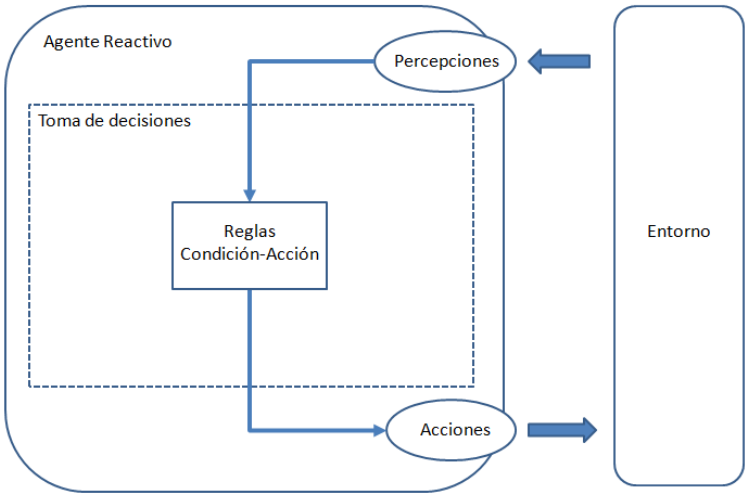


Figura 10: Arquitectura de un agente inteligente reactivo

– Agentes reactivos basados en modelo. Un agente reactivo basado en modelo es capaz de tomar decisiones en entornos no accesibles dado que el agente almacena un modelo del estado actual de los elementos no visibles del entorno. Este tipo de agentes debe ser capaz de mantener este modelo interno en función del histórico de percepciones.

  La toma de decisiones se realiza igual que en los agentes reactivos, es decir, con un conjunto de reglas condición-acción pero basadas en las percepciones y en el modelo interno del entorno.

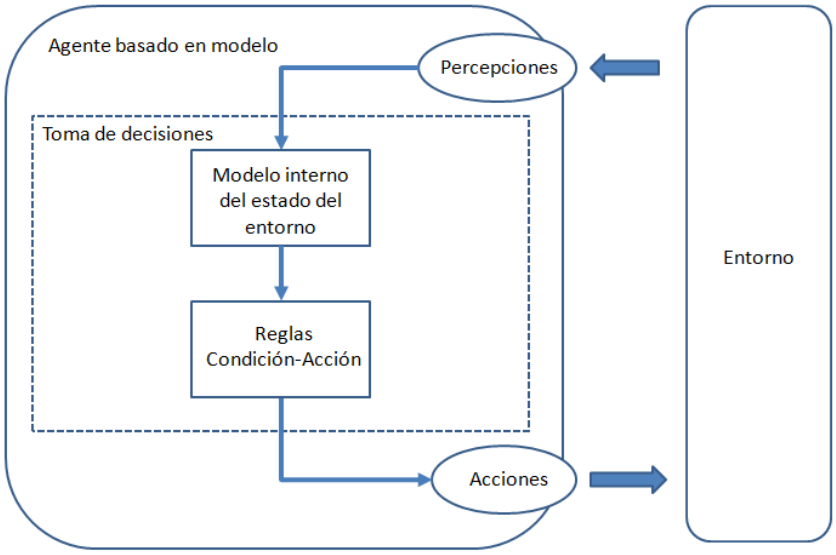


Figura 11: Arquitectura de un agente inteligente basado en modelo

– Agentes basados en objetivos. Los agentes basados en objetivos amplían las capacidades de los agentes basados en modelo al guiarse por objetivos. Los objetivos describen situaciones deseadas, esto permita al agente elegir entre un conjunto de posibilidades, seleccionando la que permite alcanzar un estado objetivo.

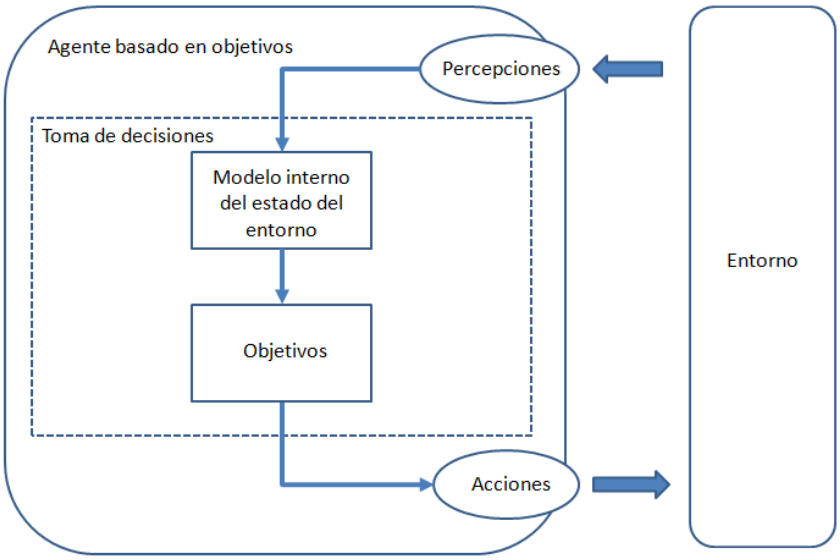


Figura 12: Arquitectura de un agente inteligente basado en objetivos

– Agentes basados en utilidad. Los agentes basados en objetivos solo distinguen estados objetivos de estados no objetivo, pero es posible definir una medición para determinar cuán deseable es un estado concreto. Esta medición se puede obtener a partir del uso de una función de utilidad que dado un estado le asigna una utilidad. De esta manera el agente es capaz de comparar los objetivos en función de su utilidad y seleccionar el de mayor utilidad.

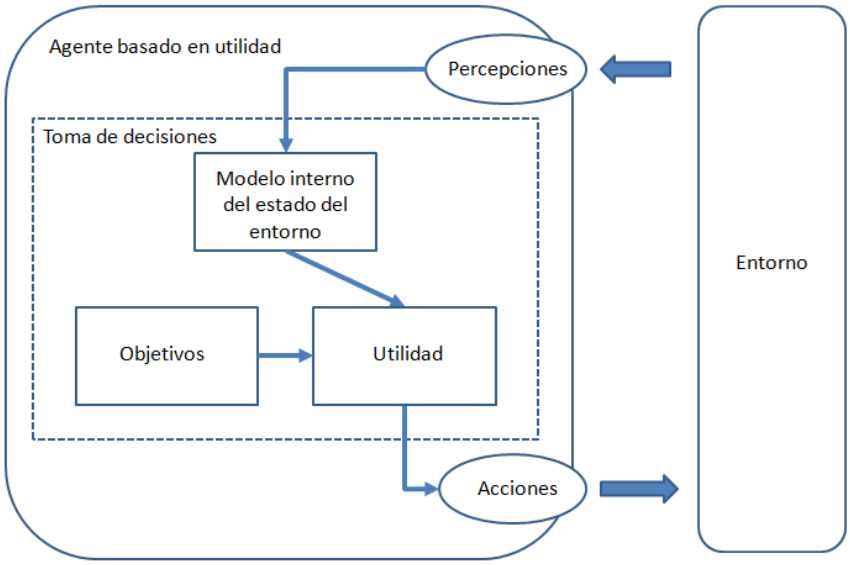


Figura 13: Arquitectura de un agente inteligente basado en utilidad

– Agentes que aprenden. El aprendizaje permite a los agentes operar en entornos desconocidos y volverse más competentes a medida que su conocimiento se amplia. Para poder aprender necesitan la ayuda de un "critico" que les indique si sus actos han sido los correctos dada una situación para poder así modificar su conducta. También tienen un componente capaz de sugerir acciones con el objetivo de permitir al agente experimentar nuevas situaciones para ampliar su conocimiento.

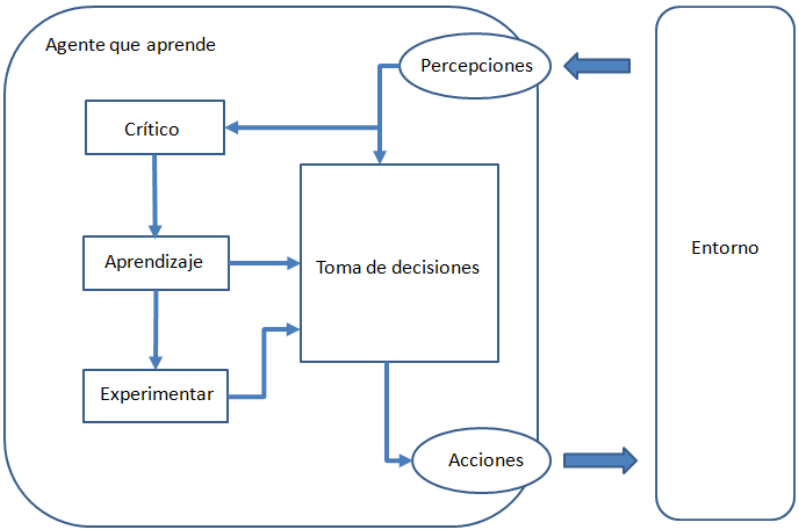


Figura 14: Arquitectura de un agente inteligente que aprende

2.2.2 Técnicas para la toma de decisiones en videojuegos

Uno de los problemas que se presentan al querer desarrollar la inteligencia artificial en videojuegos es encontrar la manera de realizar la toma de decisiones sobre qué acciones realizar lo más eficiente posible con una lógica mantenible y escalable a medida que el sistema crece en complejidad. El resultado de esta toma de decisión es el comportamiento que exhibirá el agente en un momento dado. También es deseable que los comportamientos que pueden realizar los agentes sean reutilizables, ya sea para crear comportamientos más complejos o para reutilizarlos en otros videojuegos.

Máquinas de estados

Las máquinas de estados o autómatas de estados finitos son modelos de comportamiento de un sistema con un número limitado de estados predefinidos donde existen transiciones entre estados.

Las máquinas de estados están compuestas por 4 elementos principales:

- Estados que definen el comportamiento y que producen acciones
- Transiciones de estados que son movimientos de un estado a otro

- Reglas que deben cumplirse para permitir un cambio de estado
- Eventos que permiten el lanzamiento de reglas y que permiten las transiciones

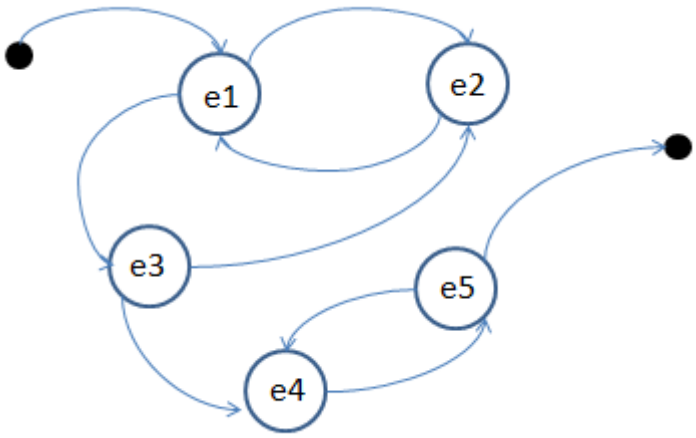


Figura 15: Máquina de estados finita

Una máquina de estados debe tener un estado inicial y un estado actual. Los eventos recibidos actúan como disparadores que causan la evaluación de las reglas que permiten la transición del estado actual a otro estado. Es posible que se activen múltiples reglas creando así un grupo de conflicto. Como solo puede haber una transición desde el estado actual, es necesaria una estrategia de resolución del conflicto para seleccionar una de las reglas activadas y así realizar la transición de estado.

Si esta estrategia consiste en seleccionar siempre la misma transición dado un conflicto, hablamos de una implementación determinista de la máquina de estados. Una máquina de estado determinista es predecible porque dadas unas entradas y un estado se puede saber qué transición se va a realizar. El problema de esta implementación en los videojuegos es que debido a que son predecibles elimina el factor de diversión del juego.

Si por el contrario la estrategia consiste en seleccionar una transición no predecible, hablamos de una implementación no determinista de la máquina de estados. Existen varias extensiones de las máquinas de estados que permiten hacer

más difícil predecir sus actos, algunas de estas opciones son el uso de lógica difusa o la selección aleatoria de las reglas.

Ventajas

- Son rápidos de diseñar, de implementar y de ejecutar.
- Son fáciles de verificar cuando son deterministas porque son predecibles, dado un grupo de entradas y un estado se puede predecir la transición de estados,
- Bajo uso del procesador.

Desventajas

- Su naturaleza predecible no los hace deseables para según qué juegos.
- Si se implementa un sistema grande puede ser difícil de administrar y mantener.
- Solo debe ser usado cuando el comportamiento de un sistema puede ser descompuesto en estados separados con condiciones bien definidas para las transiciones.
- Las reglas para las transiciones entre estados son rígidas sin el uso de lógica difusa.

Máquinas de estados jerárquicas

Uno de los principales problemas de las máquinas de estados es su baja escalabilidad. No es posible reusar lógica en diferentes contextos obligando a tener redundancia en la lógica o en tener una lógica excesivamente complicada.

Las máquinas de estados jerárquicas ayudan a la reusabilidad de dicha lógica. Mientras que en las máquinas de estado todos los estados están al mismo nivel, en las jerárquicas existen estados que engloban otros estados que comparten transiciones, permitiendo así la reusabilidad de las transiciones. Es decir, hay estados que contienen otras máquinas de estados en su interior.

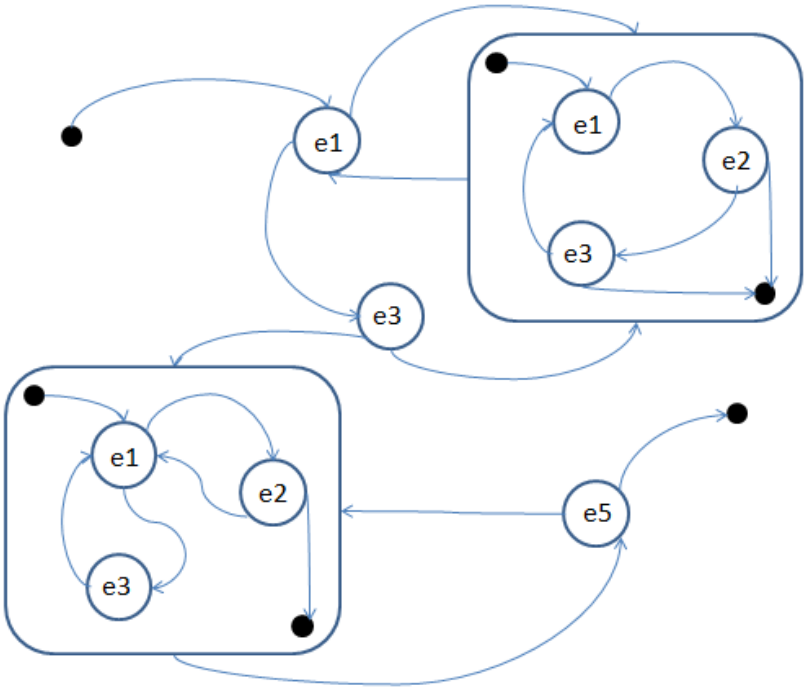


Figura 16: Maquina de estados jerárquica

Ventajas

- Permiten reutilizar transiciones.
- Permiten crear comportamientos más complejos a base de anidar otras máquinas de estados.
- Reducen la complejidad en el diseño.

Desventajas

- La depuración de la lógica es más compleja.

Árboles de comportamiento

Mientras que en las máquinas de estados, jerárquicas o no, se definen todas las posibles transiciones que hay de un estado a otro para poder decidir qué comportamiento se debe ejecutar dado el estado actual, en los árboles de comportamiento se toma un enfoque distinto.

Un árbol de comportamientos es una estructura jerárquica donde los nodos hoja son acciones y los nodos intermedios son comportamientos abstractos. El comportamiento que realiza el agente se define a medida que se recorre el árbol y se van evaluando los distintos subcomportamientos. En un nodo intermedio se definen unas precondiciones que permiten la ejecución del mismo y como consecuencia la evaluación y ejecución de uno o varios de sus hijos. La manera en cómo se evalúan y se ejecutan los hijos determina el tipo de nodo intermedio. Los tipos más habituales son los selectores, secuenciales, aleatorios y paralelos.

- Los nodos selectores evalúan los hijos hasta que uno de ellos cumple sus precondiciones y entonces lo ejecuta.
- Los nodos secuenciales evalúan y ejecutan todos sus hijos uno por uno mientras cumplan sus precondiciones.
- Los nodos aleatorios evalúan todos sus hijos y de entre todos los que cumplen sus precondiciones se ejecuta uno de ellos aleatoriamente.
- Los nodos paralelos evalúan y ejecutan sus hijos en paralelo.

Combinando las acciones que puede realizar un agente con estos nodos intermedios se pueden construir subcomportamientos que combinados entre sí producen comportamientos más complejos.

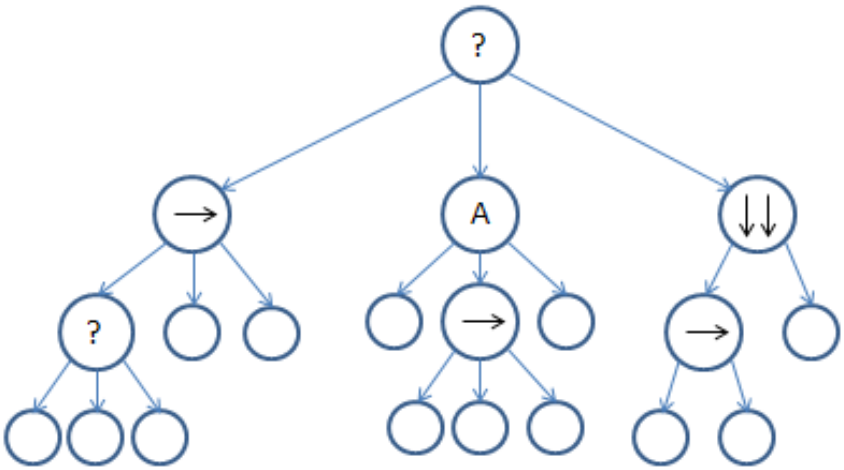


Figura 17: Árbol de comportamiento

Por lo tanto, en los árboles de comportamiento no existen estados ni transiciones sino comportamientos que se ejecutan en función del conocimiento que se tiene del entorno. Otra diferencia importante respecto a las máquinas de estados es que las máquinas de estados encapsulan los comportamientos dentro de los estados mientras que los árboles de comportamiento los hacen explícitos y reutilizables.

Ventajas

- Las precondiciones son menos numerosas que las transiciones.
- Es sencillo reutilizar la lógica en diferentes contextos.
- Su diseño es intuitivo.

Desventajas

- La depuración de la lógica es más compleja.
- Uso del procesador más elevado.

Árboles de decisión

Los árboles de decisión pueden verse como una simplificación de los árboles de comportamiento donde los nodos intermedios son nodos selectores. Como en los árboles de comportamiento, no existen ni estados ni transiciones y por lo tanto la decisión a la que se llega una vez evaluado el árbol con el conocimiento del agente es independiente de la evaluación anterior.

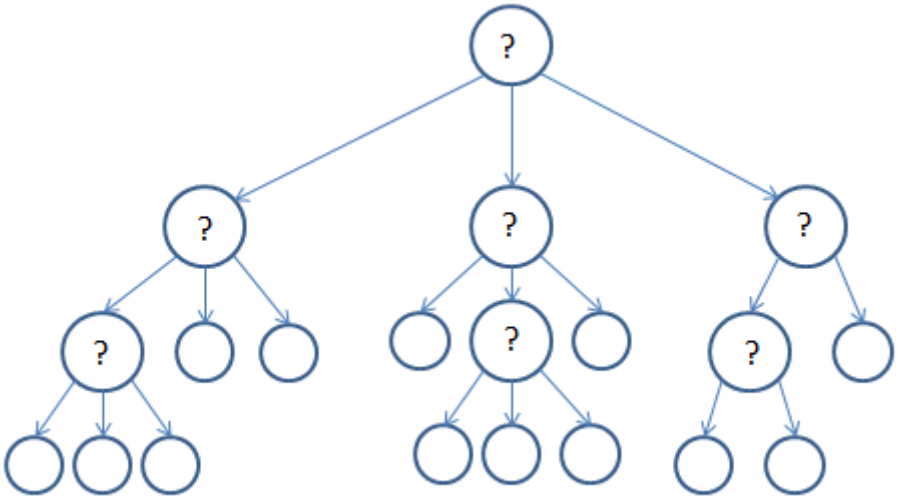


Figura 18: Árbol de decisión

Ventajas

- Las precondiciones son menos numerosas que las transiciones.
- Su diseño es intuitivo.

Desventajas

- La depuración de la lógica es más compleja.
- No permiten la reutilización de comportamientos.
- No permiten la secuenciación de comportamientos.

3 Desarrollo del motor de inteligencia artificial

3.1 Análisis

El objetivo es diseñar e implementar un motor de inteligencia artificial en tiempo real capaz de gestionar la inteligencia artificial de los agentes implicados así como de la coordinación de los mismos.

Al tratarse de un motor en tiempo real es necesario que la respuesta de la inteligencia artificial sea muy rápida, es por eso que se evitará por ejemplo el uso de técnicas como la búsqueda en el espacio de estados donde el tiempo de respuesta puede ser elevado, este tipo de técnicas es utilizado en videojuegos como los de ajedrez donde el tiempo de respuesta no tiene porque ser inmediato.

El motor estará orientado a gestionar la toma de decisiones en entornos muy dinámicos donde es complicado realizar una planificación exhaustiva de las acciones a realizar. Por lo tanto, se necesita que los agentes sean muy reactivos a los cambios del entorno pero que a la vez sean capaces de mantener sus objetivos durante un tiempo razonable. Por otro lado, se desea definir los mecanismos necesarios para que los agentes se puedan coordinar para alcanzar objetivos comunes.

Por último, se busca crear un entorno de trabajo que facilite el desarrollo y reusabilidad de la inteligencia artificial específica del problema en cuestión.

3.1.1 Requisitos funcionales

Los requisitos funcionales que debe cumplir el motor de inteligencia artificial son:

- El motor se debe ejecutar en tiempo real.
- Debe ser capaz de gestionar múltiples agentes inteligentes.
- Debe ser capaz de gestionar la coordinación entre agentes inteligentes.

3.1.2 Requisitos no funcionales

- **Eficiencia**. Es importante que las funcionalidades del motor se realicen sin un consumo excesivo de recursos.
- **Escalabilidad**. Es deseable que el diseño favorezca la incorporación de nuevas funcionalidades al motor para expandir sus posibilidades de uso.
- **Usabilidad**. Las interfaces del motor deben ser intuitivas y fáciles de usar para los desarrolladores.

3.2 Diseño

3.2.1 Agentes inteligentes

Un agente debe ser capaz de percibir los cambios que se producen en su entorno para poder decidir qué acciones debe realizar en cada momento. Un agente sólo puede tomar decisiones sobre las cosas que conoce, es por eso que es esencial la manera en cómo se representa su conocimiento.

El tipo de conocimiento puede ser muy variado, desde saber la posición del resto de agentes a saber si se encuentra lo suficientemente cerca de un objeto para interactuar con él. Un agente también debe conocer las acciones que puede realizar, que combinadas producen comportamientos, y sus objetivos tanto personales como colectivos.

Se ha realizado una distinción entre las decisiones que podrá tomar un agente basándose en su conocimiento: decisiones a largo plazo y decisiones a corto plazo.

Las decisiones a largo plazo permiten al agente decidir qué quiere o deber realizar dada una situación. Consiste en decidir cuál de los objetivos conocidos por el agente se adapta mejor a la situación actual. En cambio, las decisiones a corto plazo permiten al agente decidir cómo se va a comportar actuar para alcanzar un objetivo.

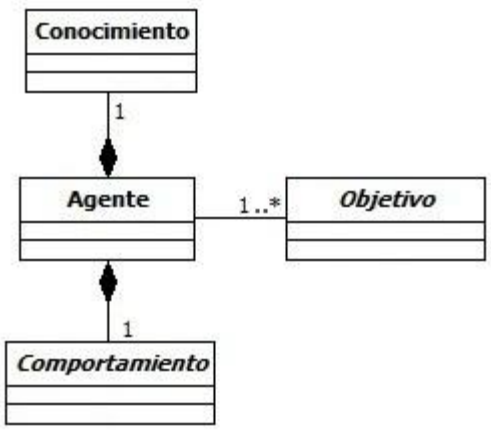


Figura 19: UML - Diagrama de clases de un agente inteligente

A continuación se muestra el diagrama de secuencia de la actualización de un agente.

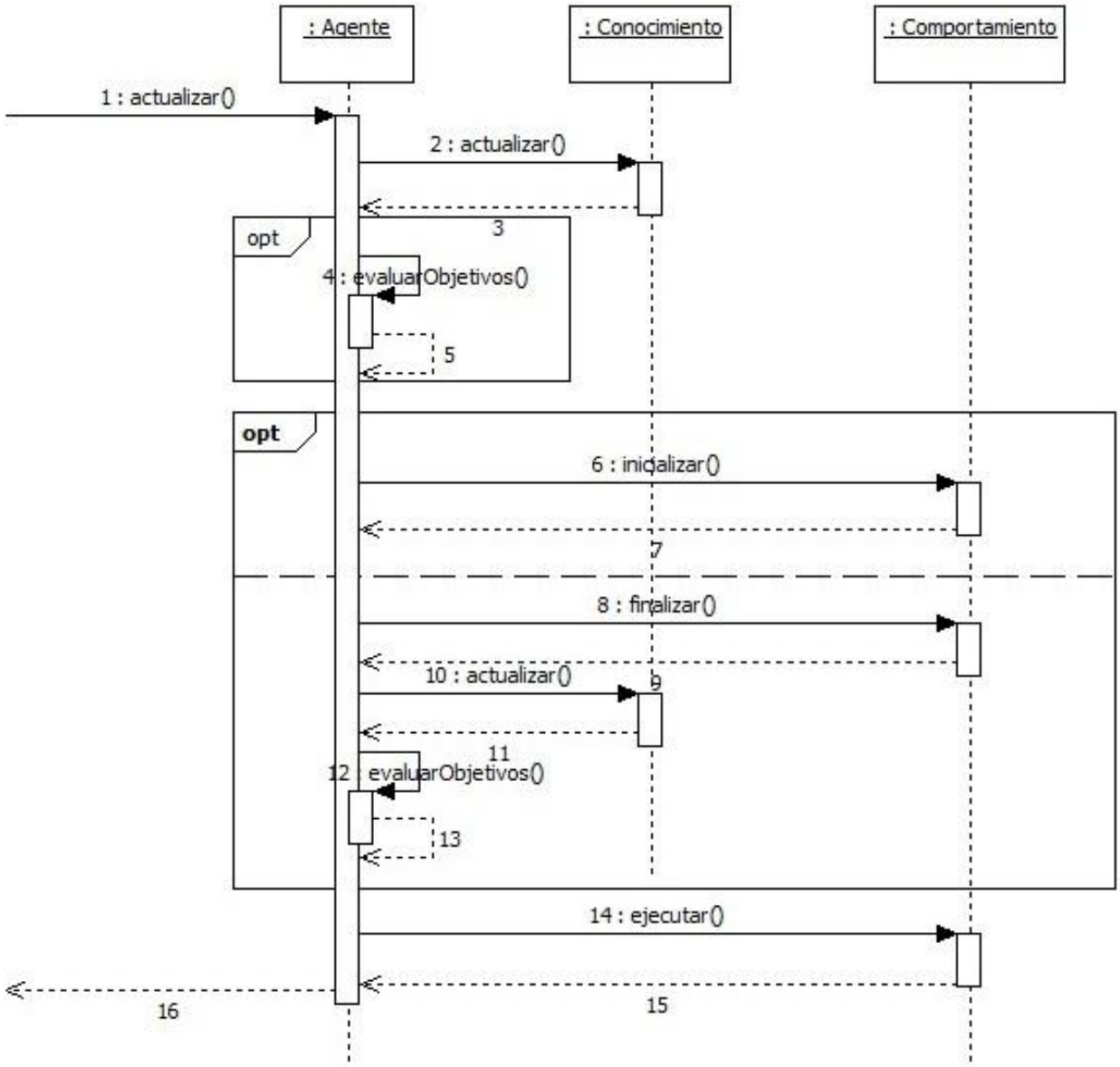


Figura 20: UML - Diagrama de secuencia de la actualización de un agente

En la figura se observa que lo primero que hace el agente es actualizar su conocimiento para poder tomar decisiones sobre el estado actual de aquello que conoce.

Para poder ser reactivo es necesario o bien estar continuamente pendiente de las cosas que ocurren en el entorno o responder sólo a eventos producidos por cambios significativos en el conocimiento. Se ha optado por la segunda solución donde al actualizar el conocimiento se produce un evento que avisa al agente de que se ha producido un cambio significativo en su conocimiento y por lo tanto sería adecuado reevaluar sus objetivos. La manera en cómo se generan dichos evento será tratada en la sección de la representación del conocimiento.

En el caso de que sea necesario comprobar si el objetivo actual del agente sigue siendo adecuado, se hace una reevaluación de los objetivos. El resultado de esta evaluación es el comportamiento que debe exhibir el agente. Por lo tanto una reevaluación del los objetivos implica modificar el comportamiento actual del agente.

Si el comportamiento actual está inactivo, resultado de una reevaluación, el agente lo inicializa. Si por el contrario el comportamiento no ha sido reemplazado pero se ha terminado de ejecutar, es necesario finalizarlo y realizar una reevaluación para poder ejecutar continuamente un comportamiento.

Por último, el agente actualiza su comportamiento actual indefinidamente hasta que este se finalice o sea necesario una reevaluación de los objetivos.

3.2.2 Conocimiento

El conocimiento representa todo aquello que conoce el agente. Si algo no existe en el conocimiento del agente, éste es incapaz de saber de su existencia.

La manera en cómo se representa dicho conocimiento es muy dependiente del problema al que se quiere aplicar el motor de inteligencia artificial por lo que uno de los primeros pasos al querer crear un agente es definir su conocimiento y cómo se va a representar.

Dado su carácter dependiente del problema, el motor ofrecer una clase abstracta que al heredarla permite darle una implementación concreta en función del problema.

De todos modos se ha identificado conocimiento común en la mayoría de agentes que se pueden desarrollar como pueden ser la posición a la que se quieren mover o hacia donde se quieren orientar.

Se han creado un conjunto de clases llamadas "Targets" (para diferenciarlos de los objetivos del agente) que permiten definir posiciones objetivo. La idea que hay detrás de los "Targets" es poder fijar posiciones tanto estáticas como dinámicas. Las posiciones estáticas representan puntos fijos del entorno mientras que las dinámicas representan las posiciones variantes de una entidad.

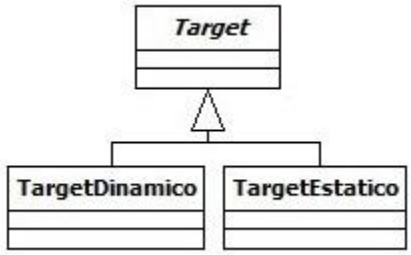


Figura 21: UML - Diagrama de clases de Targets

Para clarificar este punto, imaginemos que queremos que el agente que representa a una persona que está comprando en una tienda, se dirija hacia la puerta para salir de la tienda pero mientras se dirige a la puerta queremos que mantenga la mirada hacia una persona que se mueve perpendicularmente a él. Para ello se definiría un "TargetEstatico" con la posición de la puerta, y un "TargetDinamico" asociado a la persona que se mueve perpendicularmente de manera que la posición que representa a esa persona se actualiza automáticamente con el movimiento de la misma.

Se han definido por defecto un target para el movimiento y otro para la orientación pero pueden existir tantos targets como sean necesarios en la implementación concreta del conocimiento del agente.

Otra manera para representar el conocimiento es mediante la creación de objetos "Boleano". Estos objetos representan valores ciertos o falsos pero además tienen la

característica de que notifican el cambio de ese valor por lo que son útiles como sensores.

Si por ejemplo se quiere un sensor que avise cuando un agente se encuentra a menos de 1 metro de distancia de algún objeto, se podría crear un objeto "Boleano", el cual se actualiza con dicha comprobación en cada actualización del conocimiento, y que avisa cuando hay un cambio de esa condición. Por lo tanto generaría un aviso tanto en el momento cuando el objeto pasa a estar a menos de 1 metro como cuando pasa a estarlo más pero no mientras se mantiene dentro de los rangos.

Una manera para notificar al agente de que ha habido algún cambio significativo en el conocimiento y que puede ser necesario reevaluar sus objetivos seria mediante la comprobación de la variación de los "Boleano" que se consideren oportunos, y en el caso de que alguno de ellos haya sido modificado marcar al conocimiento como actualizado de manera que el agente tenga constancia de esta situación.

Todo agente tiene un conocimiento propio pero cuando se quiere coordinar a varios agentes es posible que compartan conocimientos por lo que el conocimiento del agente mantiene una referencia al conocimiento compartido por los agentes.

Dependiendo del problema que se quiera tratar el conocimiento compartido puede ser compartido por todos los agentes o por un grupo de agentes. La manera en cómo se gestiona dicho conocimiento no puede ser generalizado por lo que se gestionará en la implementación concreta del problema.

Por último, un agente conoce todos sus objetivos. Los objetivos se pueden agrupar de muchas maneras como por ejemplo objetivos reactivos o objetivos proactivos, no hay ninguna limitación en el numero de agrupaciones de objetivos que puede conocer un agente pero si en la evaluación de los objetivos del agente. En un momento dado un agente sólo puede evaluar una agrupación de objetivos por lo que si se quiere evaluar otra agrupación será necesario hacer un cambio dinámico de la agrupación de objetivos actual del agente.

A continuación se muestra un diagrama de clases completo de la representación genérica del conocimiento de un agente.

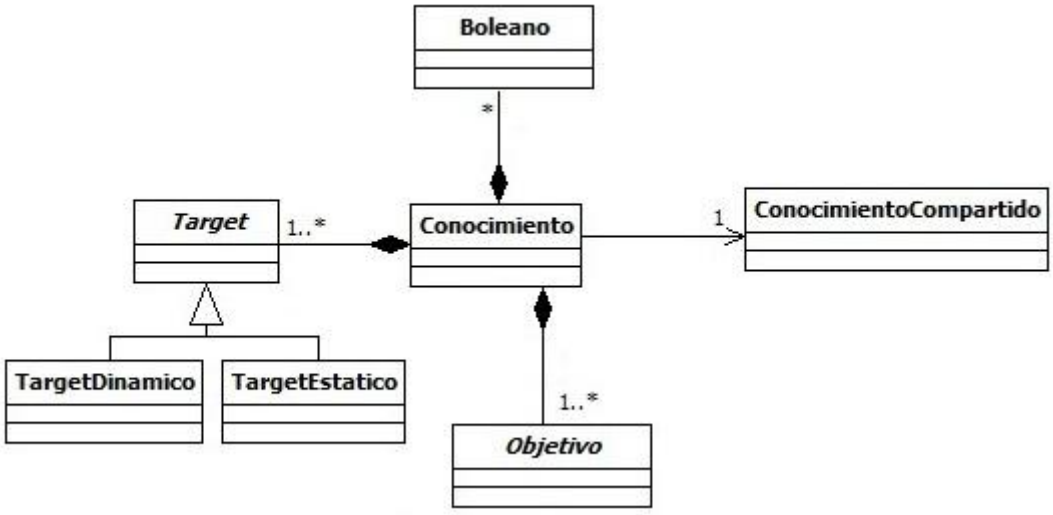


Figura 22: UML - Diagrama de clases del conocimiento de un agente

3.2.3 Toma de decisiones a largo plazo

Para la toma de decisiones a largo plazo se ha decido utilizar árboles de decisión donde los nodos representan los objetivos del agente. La decisión de qué objetivo se debe alcanzar debe perdurar en el tiempo siempre y cuando sea adecuado. No son deseable agentes que cambien constantemente de objetivo, por este motivo se le ha llamado decisiones a largo plazo.

El motivo por el cual se ha decidido utilizar los árboles de decisión es porque permiten clasificar una situación determinada de una forma sencilla y eficiente permitiendo dado el conocimiento del agente decidir cuál es el objetivo más adecuado.

Además, los arboles de decisión permiten jerarquizar de una manera sencilla los objetivos permitiendo que varios objetivos compartan las mismas precondiciones para ser seleccionados. De esta manera, en vez de comprobar cada uno de los objetivos y luego decidir cuál es el más adecuado, sólo se tienen en consideración los objetivos que cumplen las precondiciones, ignorando así objetivos no significativos.

Al ser una estructura muy intuitiva, permite a los desarrolladores definir fácilmente todos los objetivos del agente y agruparlos para reducir el número de comprobaciones necesarias para seleccionarlos.

Todos los objetivos deben tener definidas unas precondiciones que se deben cumplir para poder ser seleccionados, en el caso de que no exista una precondición indica que ese objetivo es seleccionable en cualquier situación.

A parte de dichas precondiciones, un objetivo situado en un nodo intermedio del árbol debe mantener referencias a sus sub-objetivos, mientras que si está situado en un nodo hoja mantener una referencia al comportamiento que lo implementa. A los objetivos intermedios se les ha llamado objetivos compuestos y a los nodos hoja se les ha llamado objetivos simples. Los objetivos simples pueden ser o bien objetivos individuales si pueden ser realizados por un sólo agente o objetivos colectivos si se requiere la colaboración de varios agentes para realizarlos.

Los objetivos colectivos son un tipo de objetivo especial que permiten coordinar a varios agentes y que mantienen una referencia a la táctica que lo implementa. Los objetivos colectivos serán tratados en detalle en la sección de tácticas colectivas.

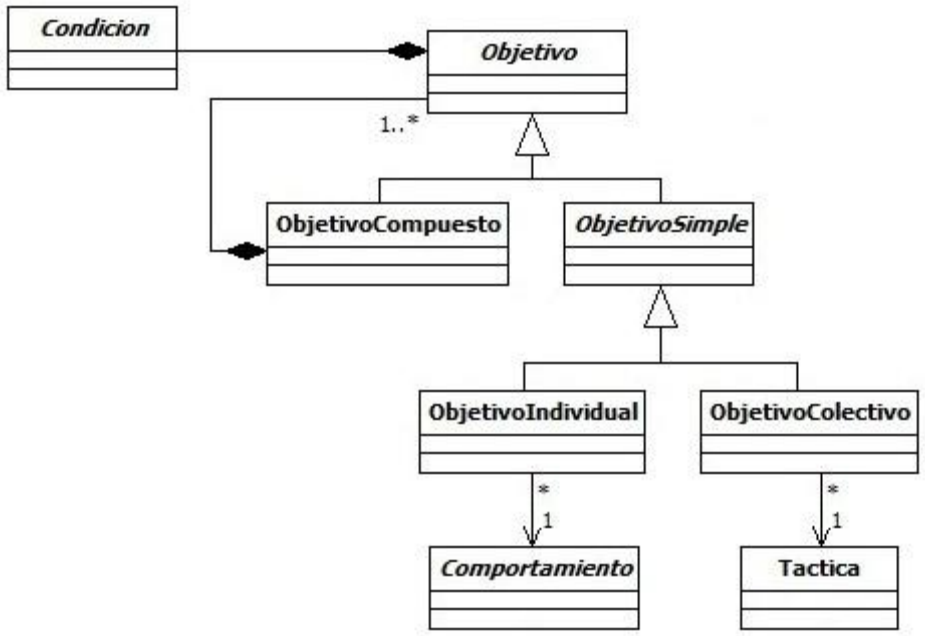


Figura 23: UML - Diagrama de clases de objetivos

El proceso de decisión de qué objetivo se debe cumplir consiste en evaluar el árbol de objetivos desde su raíz hasta que se encuentra el primer objetivo hoja que cumple con todas las precondiciones. Es por este motivo que es importante el orden de los hijos de un objetivo compuesto porque el orden determina su prioridad.

En la siguiente figura se muestra un ejemplo de evaluación de un árbol de objetivos. En ella se muestran de color amarillo los objetivos compuestos de los cuales se están evaluando sus sub-objetivos para comprobar si alguno de ellos es adecuado. Los nodos de color rojo representan los objetivos cuyas precondiciones no se cumplen y por lo tanto no son seleccionables y a su vez tampoco lo son sus sub-objetivos, los cuales están marcados de color gris.

Si tras evaluar todos los sub-objetivos de un objetivo compuesto ninguno de ellos ha podido ser seleccionado entonces el objetivo compuesto resulta fallido y por lo tanto se evaluarían sus hermanos.

Por último, el nodo de color verde representa un objetivo que cumple todas las precondiciones y por lo tanto es el objetivo seleccionado.

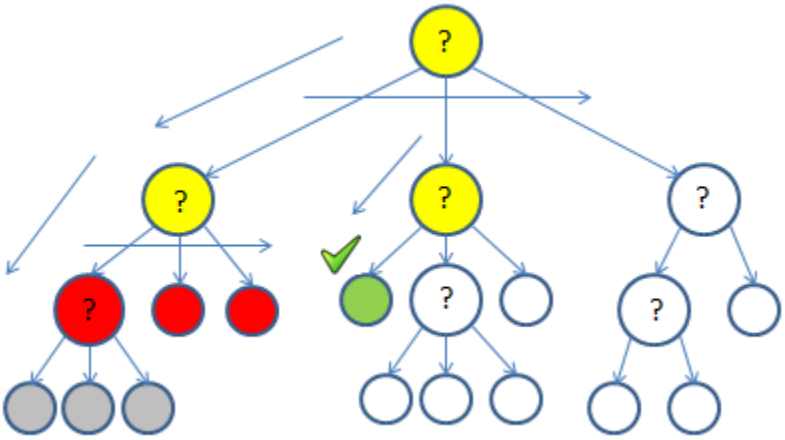


Figura 24: Evaluación de un árbol de objetivos

Como se puede observar en la figura, el orden de evaluación de los sub-objetivos es de izquierda a derecha lo cual permite de una forma sencilla definir una prioridad entre los sub-objetivos.

El siguiente diagrama de secuencia muestra cómo se realiza la evaluación del árbol de objetivos por parte de un agente para poder decidir qué comportamiento debe ejecutar.

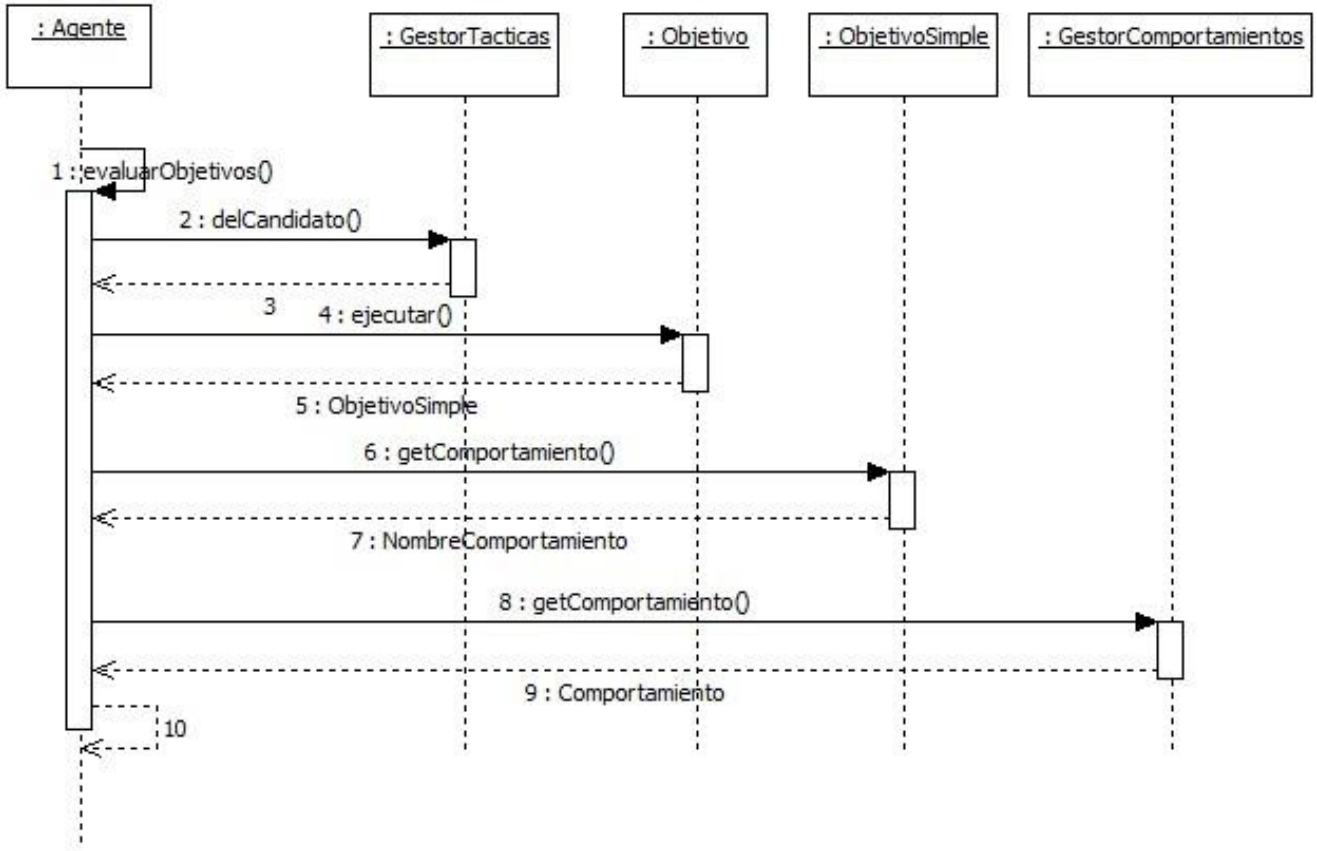


Figura 25: Diagrama de secuencia de la evaluación de un árbol de objetivos

Lo primero que hace un agente al reevaluar su árbol de objetivos es decirle al gestor de tácticas que deja de ser candidato en todas aquellas tácticas en las que esté registrado (Esto se verá en detalle en la sección de tácticas colectivas). El siguiente paso es ejecutar el árbol de objetivos para que se inicie el recorrido por el árbol para encontrar el primer objetivo simple que cumple todas sus precondiciones. Una vez se ha obtenido dicho objetivo, el agente obtiene la referencia al comportamiento que lo implementa y le pide al gestor de comportamientos que le proporcione dicho comportamiento. El comportamiento proporcionado por el gestor de comportamientos se convierte en el nuevo

comportamiento que va a ejecutar el agente hasta la próxima reevaluación del árbol de objetivos.

3.2.4 Toma de decisiones a corto plazo

Para la toma de decisiones a corto plazo se ha tomado la decisión de utilizar árboles de comportamientos. Un árbol de comportamientos es una técnica relativamente reciente que permite crear comportamientos reusables lo cual permite acelerar el desarrollo de inteligencia artificial tanto en el desarrollo actual como en el futuro.

A diferencia de los árboles de objetivos, donde la evaluación del árbol permite seleccionar el primer objetivo que cumple todas sus precondiciones, los árboles de comportamiento modifican el comportamiento que exhibe un agente a medida que se va evaluando y este puede ser evaluado indefinidamente.

Permiten modelar los comportamientos de los agentes de una manera reusable y escalable combinando varios árboles de comportamiento para crear comportamientos más complejos.

Un comportamiento se entiende por el conjunto de acciones que realiza un agente para modificar el estado de su entorno aunque también se van a considerar como acciones todas las operaciones que realiza un agente para modificar su conocimiento. Es decir, se considera una acción el proceso de realizar un paso para avanzar, así como el proceso de modificar el conocimiento con la posición a la que se quiere avanzar. La manera en cómo se ejecutan dichas acciones es dependiente del problema y por lo tanto dado un árbol de comportamiento que se quiera reutilizar para otro videojuego será necesario adaptar las acciones a la implementación concreta de dicho videojuego.

Crear un comportamiento consiste en agrupar y ordenar las acciones que puede realizar un agente de manera que al ser ejecutadas producen el comportamiento deseado. Este proceso de agrupar y ordenar se hace mediante los llamados comportamientos compuestos.

El motor va a ofrecer las cuatro implementaciones más comunes de comportamientos compuestos que se pueden necesitar a la hora de crear un árbol de comportamientos como son los selectores, las secuencias, los loops o bucles y los paralelos.

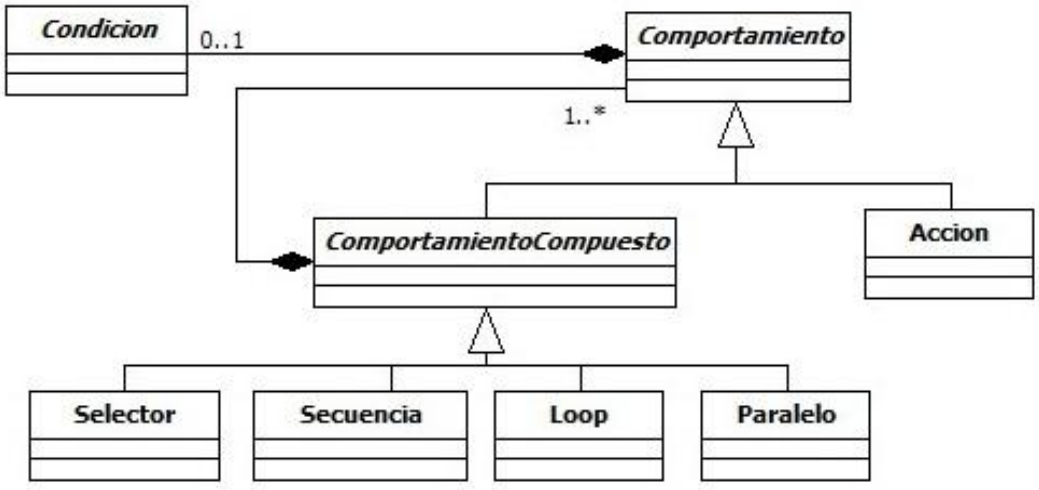


Figura 26: UML - Diagrama de clases de los comportamientos

Cada uno de ellos tiene un comportamiento diferente en función del éxito o fracaso en la ejecución de sus sub-comportamientos. Todo comportamiento va a tener cuatro posibles estados:

- **Inactivo**: el comportamiento no se ha inicializado
- **Activo**: el comportamiento está siendo ejecutado
- **Finalizado**: el comportamiento ha sido finalizado con éxito
- **Fallido**: el comportamiento no se ha podido ejecutar con éxito.

A continuación se describe la funcionalidad de cada implementación de comportamiento compuesto que ofrece el motor.

- **Selectores.** Si durante la evaluación de un nodo selector, el sub-comportamiento que se está ejecutando cambia a un estado Finalizado entonces el nodo selector también cambia su estado a Finalizado

completando su ejecución. Si por el contrario, el sub-comportamiento cambia su estado a Fallido entonces el nodo selector ejecuta el siguiente sub-comportamiento. Si el nodo selector se queda sin sub-comportamientos a los que ejecutar entonces cambia su estado a Fallido.

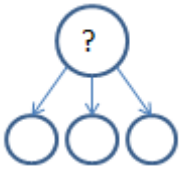


Figura 27: Comportamiento selector

- **Secuencias.** Si durante la evaluación de un nodo secuencia, el sub-comportamiento que se está ejecutando cambia su estado a Finalizado entonces el nodo secuencia ejecuta el siguiente sub-comportamiento. Si por el contrario, el sub-comportamiento cambia su estado a Fallido entonces el nodo secuencia cambia su estado a Fallido y termina su ejecución. Si el nodo selector se queda sin sub-comportamientos a los que ejecutar entonces cambia su estado a Finalizado y terminando su ejecución con éxito.

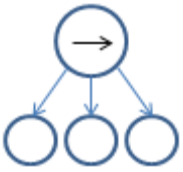


Figura 28: Comportamiento secuencia

Loops. Un nodo "loop" ejecuta indefinidamente un sub-comportamiento siempre y cuando cumpla sus precondiciones. Si el sub-comportamiento finaliza, el nodo "loop" lo vuelve a inicializar y a ejecutar.

Figura 29: Comportamiento loop

- **Paralelo.** Un nodo paralelo ejecuta paralelamente todos sus sub-comportamientos. Si alguno de sus sub-comportamientos cambia su estado a Fallido entonces el nodo paralelo cambia su estado también a Fallido terminando así su ejecución. Si todos sus sub-comportamientos cambian su estado a Finalizado entonces el nodo paralelo cambia su estado a Finalizado terminando la ejecución con éxito.

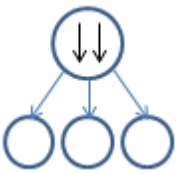


Figura 30: Comportamiento paralelo

Mientras que las acciones en un árbol de comportamientos son dependientes del problema, los comportamientos compuestos son genéricos y por lo tanto reutilizables en múltiples soluciones.

Para facilitar la tarea de reusar los árboles de comportamiento, el motor de inteligencia artificial ofrece un gestor de comportamientos cuya finalidad es almacenar todos los comportamientos y que permitir obtenerlos a partir de su nombre.

Figura 31: UML - Diagrama de clases del gestor de comportamientos

Dado que el número de comportamientos puede ser muy elevado, por cuestiones de eficiencia se ha decidido que el gestor no almacene instancias de árboles de comportamientos sino que cuando se pide un comportamiento éste se genere dinámicamente, reduciendo así la cantidad de memoria necesaria para gestionar los comportamientos. Los comportamientos se obtienen por nombres representados mediante un tipo enumerado.

3.2.5 Condiciones

Las precondiciones que debe cumplir un nodo para poder ser seleccionado pueden ser muy variadas, desde una sola condición a la combinación de varias. Para poder mantener los nodos lo más genéricos posibles, es deseable poder asignar estas precondiciones de manera dinámica.

Es por ese motivo que se han creado las condiciones que representan los operadores lógicos habituales "and", "or" y "not".

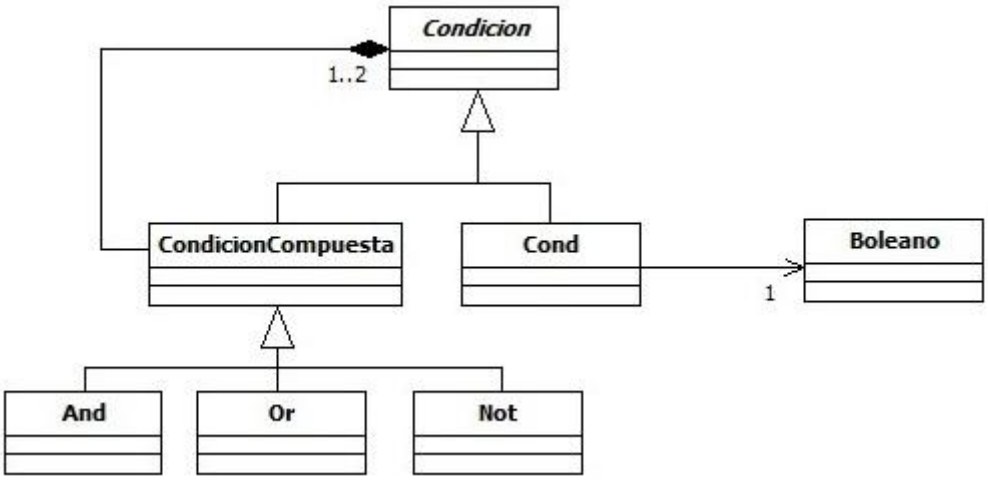


Figura 32: UML - Diagrama de clases de las condiciones

En la figura se puede observar que existe un operador llamado "cond", su finalidad es mantener una referencia a un boleano que representa un valor cierto o falso y actuar como operador identidad. El resto de operadores se combinan entre sí para poder crear condiciones más complejas.

Para clarificar la creación de las condiciones veamos el siguiente ejemplo en el que un soldado tiene que decidir si puede disparar su arma o no. Para poder disparar el arma es necesario que el arma tenga munición y haya enemigos a la vista pero que no tenga un enemigo acercándose por la espalda.

Para poder representar dichas condiciones es necesario crear primero tres booleanos que nos indiquen si el arma tiene munición, si hay enemigos a la vista y si hay alguien acercándose por la espalda. Para ello se pueden definir los siguientes booleanos:

- tiene_municion
- enemigos_visibles
- en_peligro

El siguiente paso es combinar dichos booleanos para crear la precondición de la siguiente manera:

```
And(Cond(tiene_municion),And(Cond(enemigos_visibles),Not(Cond(en_peligro))))
```

La siguiente figura muestra cómo se evalúa una condición de tipo “cond”:

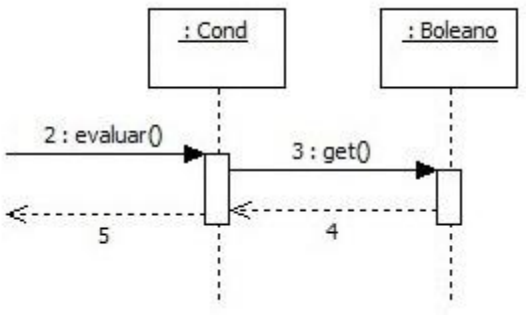


Figura 33: UML - Diagrama de secuencia de la evaluación de una condición

El resto de operadores evalúan sus condiciones asociadas y aplican el operador lógico que representan al resultado de esas evaluaciones.

3.2.6 Tácticas colectivas

En algunos problemas es necesario que los agentes se coordinen entre sí para poder solucionarlos o encontrar mejores soluciones que de otra manera les sería imposible o muy complicado. Por lo tanto, el motor debe ofrecer mecanismos para que los agentes sean capaces de coordinarse. Se han diferenciado dos tipos de coordinación: la explícita y la implícita.

3.2.6.1 Coordinación explícita

La coordinación explicita se entiende por la coordinación mediante cualquier tipo de comunicación entre los agentes. Por ejemplo, los agentes se pueden comunicar mediante un espacio de memoria compartido como puede ser el conocimiento compartido que ofrece el motor de inteligencia artificial. Esta técnica es conocida como arquitectura en pizarra donde los agentes examinan la pizarra, realizan su tarea y modifican la pizarra. De esta manera, otro agente puede trabajar sobre los resultados generados por otro. Por lo tanto permite coordinar a los agente y facilita su intercomunicación.

La información necesaria para coordinar los agentes mediante el conocimiento compartido es dependiente del problema que se quiere resolver y por lo tanto el motor sólo ofrece una interfaz común que cada videojuego deber implementar con sus características propias.

Sin embargo, en ocasiones es necesario limitar el número de agentes necesarios para realizar una tarea e incluso asignar un rol diferente dentro de la coordinación en función de las características de los agentes o el estado del entorno. Para gestionar este tipo de coordinación el motor ofrece la creación de tácticas integradas en los árboles de objetivos.

Una táctica define una coordinación entre agentes para solucionar un problema donde existe un mínimo y un máximo de agentes que pueden participar en ella así como un mecanismo para seleccionar a los mejores candidatos y detectar cuando la táctica ha sido realizada con éxito o si por el contrario ya no es adecuada.

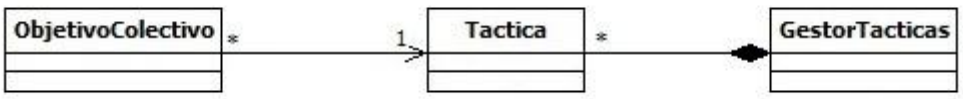


Figura 34: UML - Diagrama de clases del gestor de tácticas

El motor ofrece un tipo de objetivo especial llamado objetivo colectivo que permite a los agentes participar en las tácticas colectivas. Los objetivos colectivos en vez de mantener una referencia a un comportamiento, mantienen una referencia a una táctica gestionada por el gestor de tácticas.

Cuando un agente evalúa su árbol de objetivos y se alcanza un objetivo colectivo que cumple con todas las precondiciones, entonces se comprueba si el agente es apto para participar en la táctica que implementa dicho objetivo colectivo. Si el agente es apto pero no hay suficientes agentes para realizar dicha táctica entonces se marca al agente como candidato pero no se le permite ejecutar la táctica con lo cual el agente debe buscar otro objetivo en su árbol de objetivos.

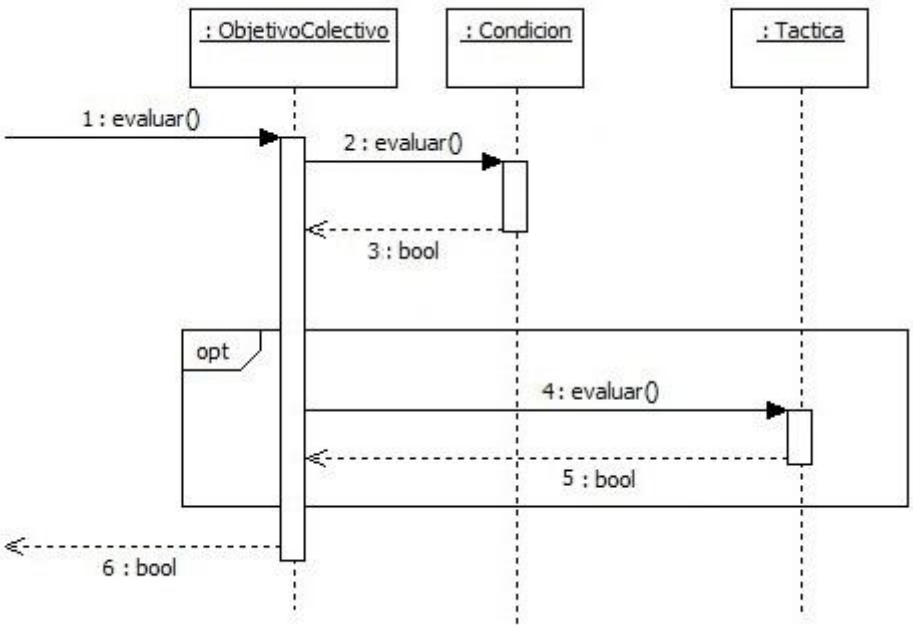


Figura 35: UML - Diagrama de secuencia de la evaluación de un objetivo colectivo

Cuando una táctica tiene los candidatos suficientes para iniciarla entonces le pide a todos sus agentes candidatos que reevalúen sus respectivos árboles de objetivos para verificar que siguen siendo candidatos para realizara. Si tras esta comprobación todos los candidatos son aptos, entonces empiezan a ejecutar dicha táctica colectiva hasta que se termina con éxito o deja de ser adecuada.

Si mientras se está realizando una táctica otros agentes se ofrecen a participar en ella, la táctica puede comprobar si el nuevo candidato es mejor que alguno de los que están participando en ella y reemplazarlo.

Por último, las tácticas comprueban si se ha realizado con éxito la coordinación o por el contrario ha dejado de ser adecuada ya sea porque se ha quedado sin candidatos suficientes para terminarla o ha dejado de tener sentido dentro del contexto del problema que se quiere resolver. En este caso es necesario que todos los agentes registrados en la táctica reevalúen sus respectivos árboles de objetivos.

A continuación se muestra el diagrama de secuencia que muestra la comprobación de las tácticas por parte del gestor de tácticas.

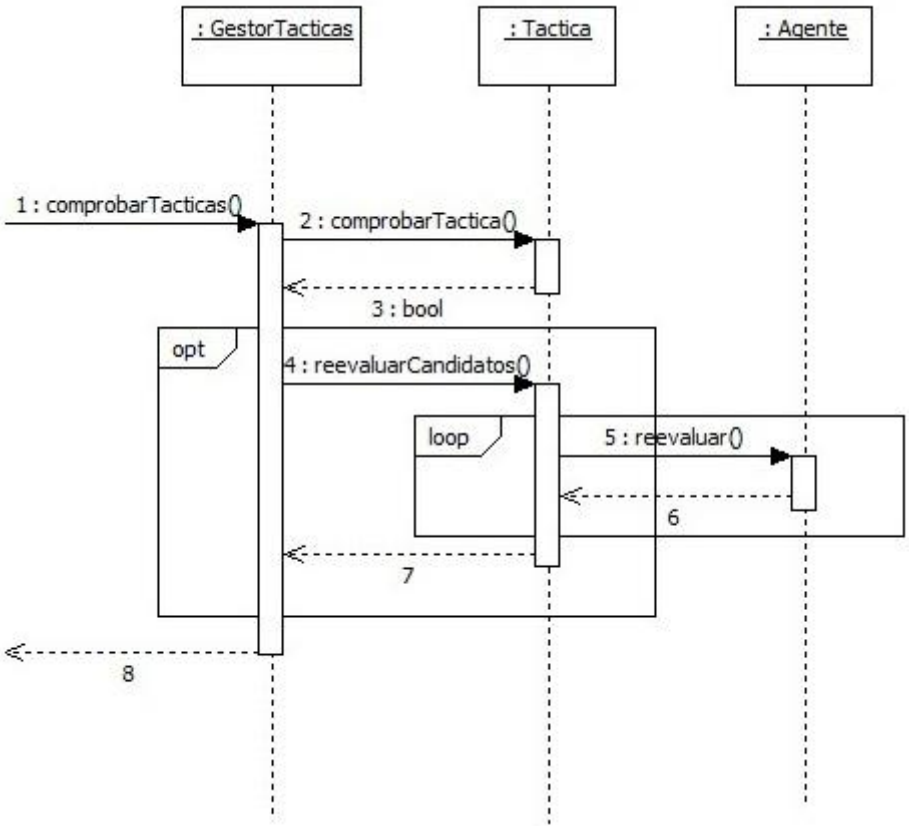


Figura 36: UML - Diagrama de secuencia de la comprobación de la viabilidad de un táctica colectiva

3.2.6.2 Coordinación implícita

La coordinación implícita se entiende por la coordinación donde no es necesaria la comunicación o incluso sería contraproducente porque podría dar pistas al adversario. Este tipo de coordinación se obtiene a partir de la observación de las acciones que realizan el resto de agentes.

Un ejemplo de coordinación implícita sería el movimiento en formaciones por parte de los agentes. El objetivo de las formaciones es mantener una posición relativa respecto al resto de agentes.

Este tipo de coordinación se puede obtener mediante la creación de comportamientos que tengan en cuenta al resto de agentes y donde no es necesario realizar ningún tipo de notificación. Por este motivo el motor de inteligencia artificial no ofrece ningún mecanismo especial para tratar este tipo de coordinación salvo la creación de los árboles de comportamientos.

3.2.7 Motor de inteligencia artificial

El objetivo final del motor de inteligencia artificial es poder gestionar toda la inteligencia artificial de un videojuego separando la lógica del videojuego de la lógica necesaria para desarrollar la inteligencia artificial.

Para dotar a una entidad de inteligencia artificial es necesario asociarle un agente inteligente con un conocimiento concreto y notificar al motor de inteligencia artificial de su existencia para que a partir de entonces se encargue de su gestión.

A parte de gestionar los agentes que participan en el videojuego también se tiene que encargar de actualizar el conocimiento que es compartido por todos los agentes así como supervisar las tácticas colectivas que se están llevando a cabo.

Para facilitar la creación de árboles de comportamiento, el motor de inteligencia artificial mantiene referencias a los gestores de comportamiento y de tácticas para que sean accesibles desde cualquier punto del motor.

A continuación se muestra el diagrama de clases de los elementos gestionados directamente por la clase MotorIA cuya función es hacer de interfaz del motor con el exterior.

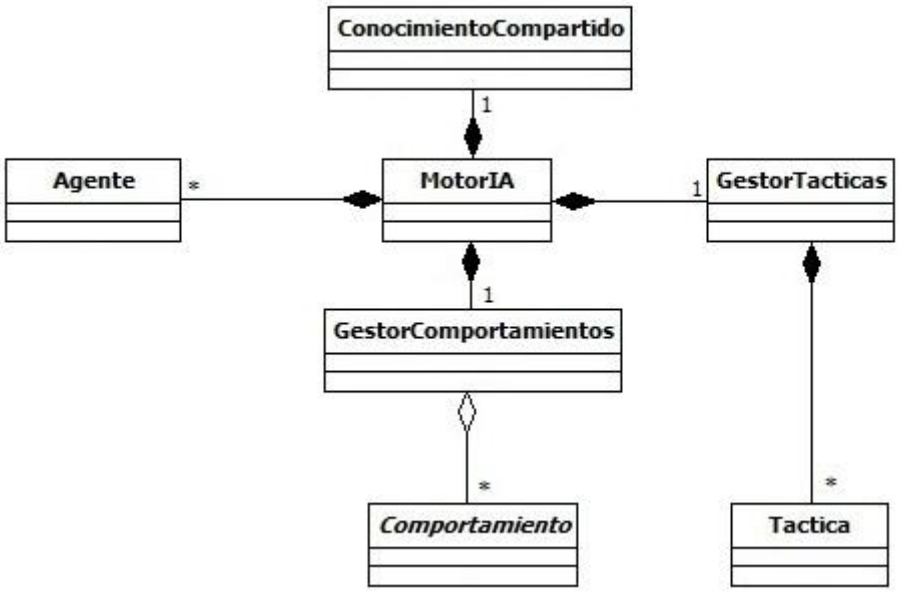


Figura 37: UML - Diagrama de clases del motor de inteligencia artificial

La siguiente figura muestra el diagrama de secuencia de la creación del motor y sus principales componentes.

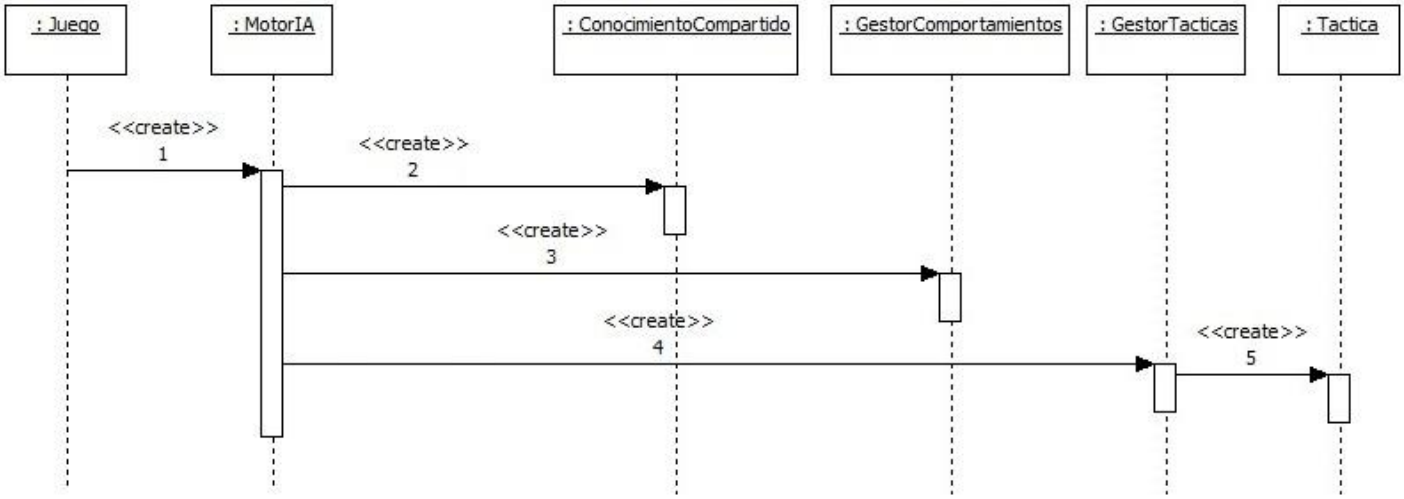


Figura 38: UML - Diagrama de secuencia de la creación del motor de inteligencia artificial

A continuación se muestra el diagrama de secuencia de la inicialización del motor una vez se han creado todos los agentes que participan en el videojuego.

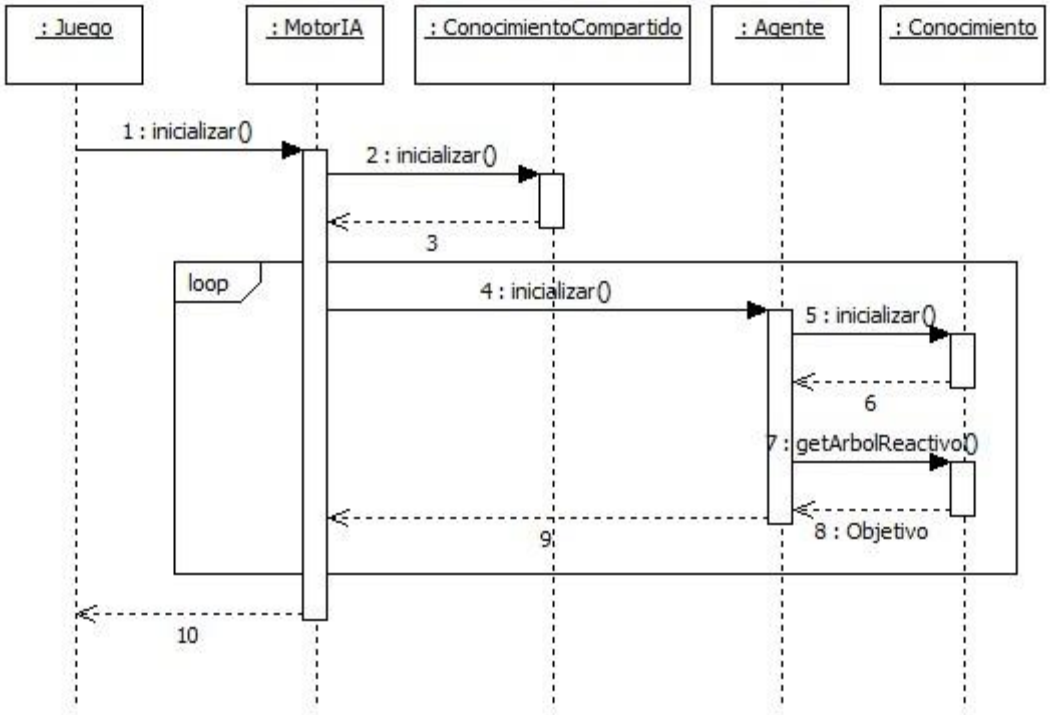


Figura 39: UML - Diagrama de secuencia de la inicialización del motor de inteligencia artificial

Se puede observar que durante la inicialización de los agentes se obtiene el árbol de objetivos del conocimiento del agente que va a ser evaluado en primera instancia.

La siguiente figura muestra el diagrama de clases completo del motor de inteligencia artificial para poder tener una visión completa del mismo.

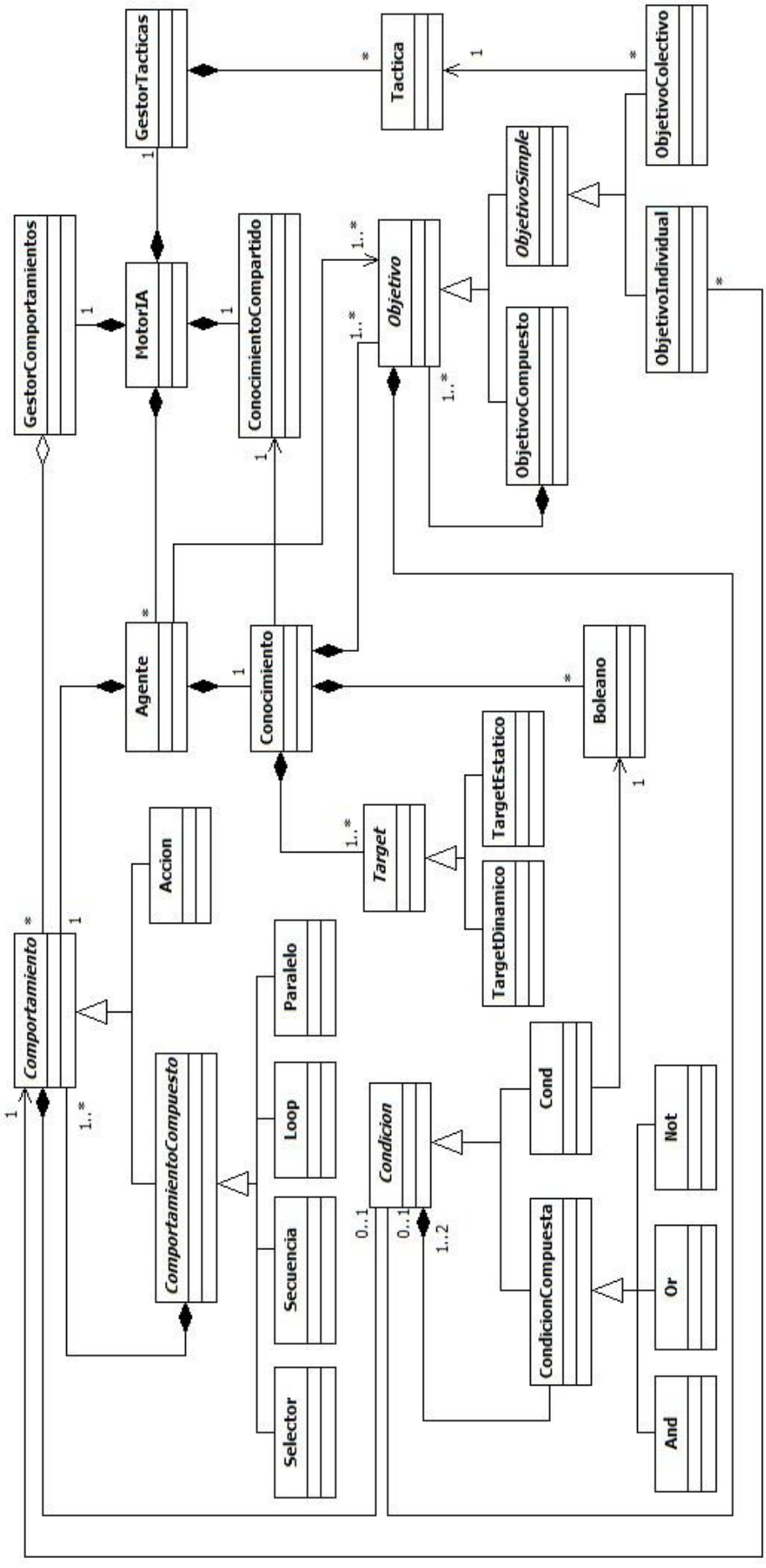


Figura 40: UML - Diagrama de clases completo del motor de inteligencia artificial

3.2.8 Patrones de diseño

Composite

El patrón Composite sirve para construir objetos complejos a partir de otros más simples o similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados ya que al poseer todos una interfaz común se tratan todos de la misma manera.

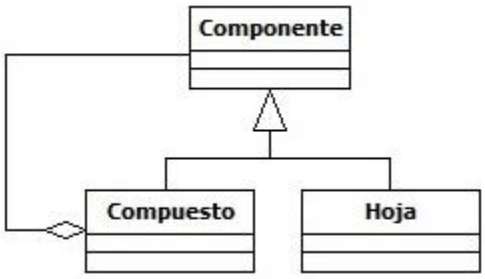


Figura 41: Patrón de diseño "Composite"

Este patrón ha sido utilizado en varias ocasiones en el motor de inteligencia artificial. Los árboles de objetivos, los árboles de comportamientos y las condiciones hacen uso de este patrón para poder generar estructuras jerárquicas.

Fachada

El patrón Fachada se utiliza para proporcionar una interfaz unificada de alto nivel para un conjunto de clases en un subsistema, haciéndolo más fácil de usar. Simplifica el acceso a dicho conjunto de clases ya que se realiza a través de una única interfaz.

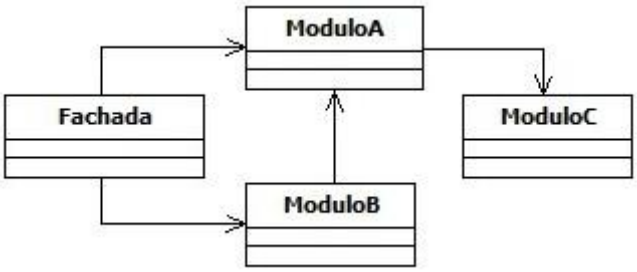


Figura 42: Patrón de diseño "Fachada"

Para crear una interfaz única del motor se ha creado la clase MotorIA aplicando el patrón Fachada.

Singleton

El patrón de diseño Singleton está diseñado para restringir la creación de objetos pertenecientes a una clase. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Figura 43: Patrón de diseño "Singleton"

Todos los gestores del motor de inteligencia artificial son clases Singleton porque sólo debe existir una instancia para cada uno de ellos y además se requiere que sean accesibles desde cualquier punto del motor para poder llamar a sus métodos.

Blackboard

El patrón Blackboard es un patrón de arquitectura de software que permite a múltiples agentes leer y modificar información compartida para comunicarse entre ellos.

Como solución a la coordinación explicita de los agentes que participan en el videojuego se ha decidido aplicar el patrón BlackBoard para crear el conocimiento compartido por todos los agentes.

4 Desarrollo del videojuego

El objetivo es desarrollar un videojuego de fútbol donde poder integrar el motor de inteligencia artificial que se quiere implementar para poder demostrar su correcto funcionamiento. Al no ser el objetivo principal de este proyecto se ha decidido desarrollar un videojuego lo más sencillo posible que permita al usuario interactuar con él y que muestre el resultado de la ejecución del motor de inteligencia artificial.

Para simplificar el desarrollo del videojuego se ha decidido utilizar un motor de físicas y un motor gráfico de código libre y no desarrollar los subsistemas de redes ni de sonido. Así mismo, tampoco se dará énfasis en el desarrollo de contenidos del videojuego.

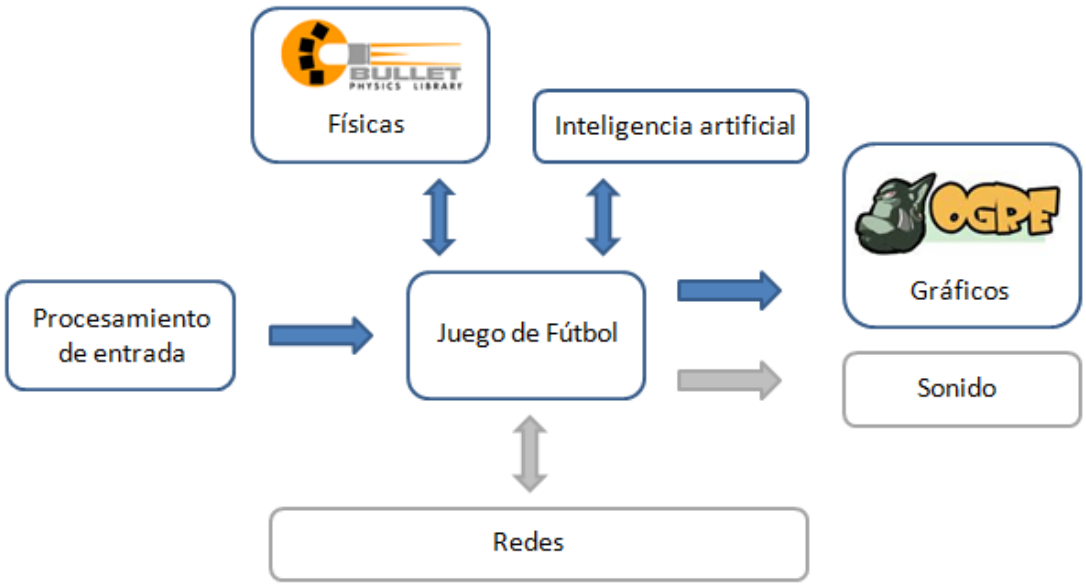


Figura 44: Diagrama de los subsistemas del videojuego implementados

Requisitos

- Visualización de todos los futbolistas para poder ver los movimientos en conjunto.
- Visualización del estado de la inteligencia artificial.
- Interacción simple del usuario para poder controlar a los futbolistas.

4.1 Tecnologías

4.1.1 C++

C++ es un lenguaje de propósito general que combina características de lenguajes de alto y bajo nivel. Su base se encuentra en el lenguaje C con el cual mantiene compatibilidad. La principal aportación de C++ es el soporta a la programación orientada a objetos. Este es un paradigma de programación que utiliza objetos (conjuntos de datos y funciones) para el diseño y estructuración de programas. C++ proporciona las siguientes funcionalidades.

- **Herencia múltiple.** La herencia permite a una clase adquirir las propiedades de otra. C++ soporta herencia múltiple que permite adquirir las propiedades de varias clases a la vez
- **Encapsulación.** Es el proceso por el cual se oculta la información con el objetivo de asegurar que las estructuras de datos y los operadores son usados correctamente. C++ consigue este propósito con la declaración de clases y funciones. En una clase lo miembros es pueden declarar públicos, protegidos o privados según se permita el acceso desde otras clases o no.
- **Polimorfismo.** El polimorfismo permite definir una interfaz común para diferentes implementaciones y que un objeto actúe de diferente manera dependiendo de las circunstancias.
- **Abstracción.** Consiste en amagar la complejidad existente en un sistema permitiendo tratarlo a un nivel de abstracción adecuado para cada tarea.

C++ es un lenguaje compilado, esto hace que en comparación con otros lenguajes interpretados como Java o C# tenga la ventaja de crear programas con un alto rendimiento. En el caso de los videojuegos es una gran ventaja ya que suelen utilizar una gran parte de los recursos del sistema. En cambio, el hecho de usar C++ obliga a compilar para un dispositivo y un sistema operativo determinado.

4.1.2 Ogre 3D

Ogre 3D (Object-Oriented Graphics Rendering Engine) es un motor gráfico 3D orientado a escenas, escrito en el lenguaje de programación C++.

Figura 45: Logotipo de OGRE 3D

4.1.2.1 Características

Plataforma

- Programado en C++.
- Soporte para Direct3D y OpenGL
- Soporte para Windows, Linux y Mac OSX

Materiales y Shaders

- Potente lenguaje para la declaración de materiales que permite mantener las características de los materiales fuera del código del programa
- Soporte para vértices y shaders de bajo y alto nivel.
- Capacidad para definir diferentes técnicas para un mismo material. Esto permite crear alternativas para facilitar la compatibilidad con tarjetas gráficas más antiguas.
- Carga de texturas en formato PNG, JPEG, TGA, BMP y DDS
- Soporte para texturas dinámicas

Modelos y animación

- Uso de un formato propio, potente y flexible

- Soporte para modelos con diferentes niveles de detalle. Pueden ser generados manualmente o automáticamente.
- Sofisticado sistema de animación mediante esqueletos
- Sistema de animación por transformación

Escena

- Sistema de gestores de escenas con gran flexibilidad y capacidad para la personalización. Da la opción de usar uno de los gestores proporcionados para los tipos de escenas más habituales o de implementar uno de propio.
- Escena organizada como una jerarquía de nodos que siguen los movimientos de sus padres.
- Diferentes técnicas de sombreado que se pueden configurar a medida y que son aceleradas según el hardware disponible.

Efectos especiales

- Soporte para efectos aplicados después de pintar la pantalla complete
- Potente y flexible sistema de partículas. Se proporciona un gran numero de emisores y modificadores y se posibilita la creación de otros personalizados
- Objetos transparentes gestionados automáticamente.

Otras funcionalidades

- Sistema de recursos para la gestión de memoria y soporte para la carga de archivos comprimidos.
- Sistema de complementos que permite extender las funcionalidades sin la necesidad de recompilar.

Su diseño y potencia ha propiciado que este motor sea una de las soluciones gratuitas más utilizadas. Dispone de una amplia documentación que cubre todas las clases que forman el motor. También están disponibles múltiples guías y demostraciones que facilitan el proceso de aprendizaje.

4.1.3 Bullet Physics

El objetivo de un motor de físicas es realizar la detección de colisiones, resolver las colisiones y otras restricciones y proveer de la actualización de todos los objetos.

Bullet es un motor de físicas de código libre que permite la detección de colisiones y de dinámica de tanto de cuerpos rígidos como de blandos.

Figura 46: Logotipo de Bullet Physics

4.1.3.1 Características

- Programado en C++ bajo la licencia Zlib y gratuito para cualquier uso comercial en todas las plataformas incluyendo: PlayStation3, Xbox 360, Wii, PC, Linux, Mac OSX y iPhone.
- Detección de colisiones discretas y continuas incluyendo rayos y barrido convexo. Los cuerpos de colisión incluyen mallas cóncavas y convexas, y todas las primitivas básicas.
- Solucionador de restricciones de dinámicas de cuerpos rígidos rápido y estable
- Dinámicas de cuerpos blandos para ropa, cuerdas y volúmenes deformables.
- Complemento para Maya, integración con Blender y soporte para importación/exportación de COLLADA.

Bullet ha sido diseñado para ser configurable y modular y se puede usar de las siguientes maneras:

- Usar solo el componente de detección de colisiones

- Usar el componente de dinámicas para cuerpos rígidos sin el componente de dinámicas de cuerpos blandos.
- Usar pequeñas partes y extenderlas.
- Elegir entre precisión simple o doble.
- Configurar las opciones de depuración.

Los principales componentes están organizados de la siguiente manera:

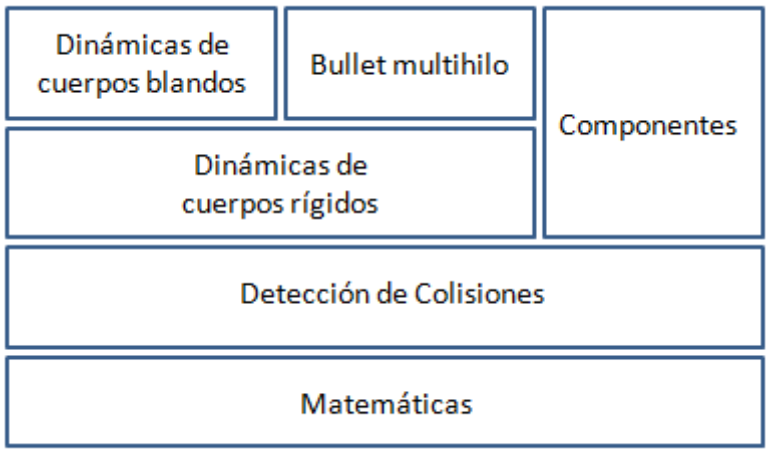


Figura 47: Arquitectura de Bullet Physics

4.1.4 Herramientas

4.1.4.1 Blender

Blender es un programa multiplataforma dedicado al modelado, animación y creación de gráficos tridimensionales.

Figura 48: Logotipo de Blender

4.1.4.1.1 Características

- Multiplataforma, libre, gratuito y de poco tamaño.
- Capacidad para crear una gran variedad de primitivas geométricas, incluyendo curvas, mallas poligonales, vacíos.
- Junto a las herramientas de animación se incluyen cinemática inversa, deformaciones por armadura o cuadrícula, vértices de carga y partículas estáticas y dinámicas.
- Edición de audio y sincronización de video.
- Características interactivas para juegos como detección de colisiones, recreaciones dinámicas y lógica.
- Posibilidades de pintado interno versátil e integración externa con potentes trazadores de rayos.
- Lenguaje Python para automatizar o controlar varias tareas.
- Acepta formatos gráficos como TGA, JPG, Iris, SGI o TIFF.
- Motor de juegos 3D integrado, con un sistema de bloques lógicos. Para más control se usa programación en lenguaje Python.
- Simulaciones dinámicas para cuerpos blandos, partículas y fluidos.
- Modificadores apilables para la aplicación de transformación no destructiva sobre mallas.
- Sistema de partículas estáticas para simular cabellos y pelajes.

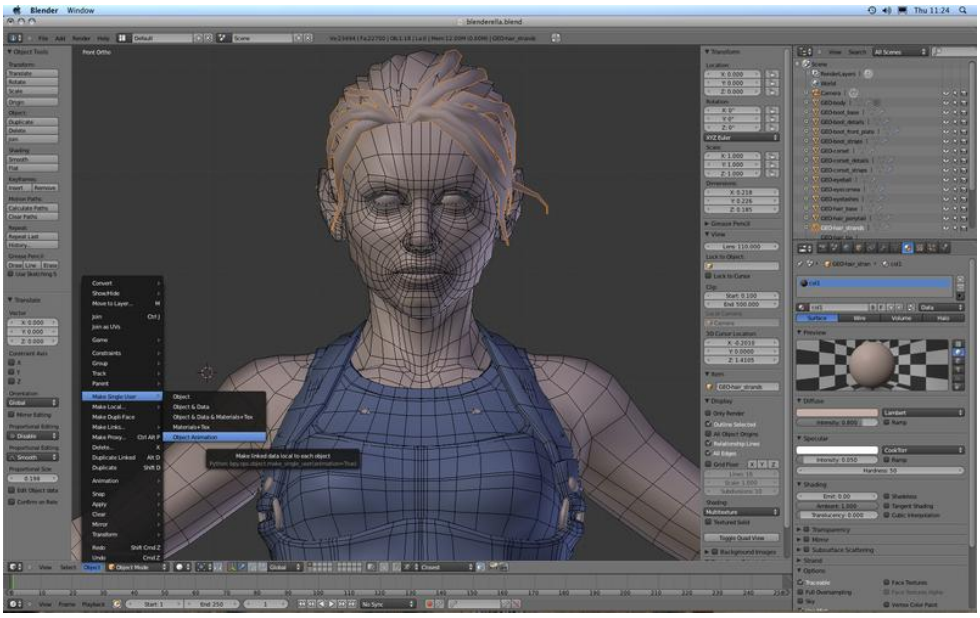


Figura 49: Captura de pantalla de Blender

4.1.4.1.2 Ogre Meshes Exporter

Ogre 3D utiliza un sistema propio para guardar la geometría y las características de los materiales. Por lo tanto, para conseguir que los modelos creados en Blender puedan ser utilizados por el motor gráfico Ogre 3D es necesario realizar una exportación que haga la traducción del formato original de Blender al formato que utiliza Ogre 3D. Para realizar dicha exportación es necesario instalar en Blender el compoenente Ogre Meshes Exporter.

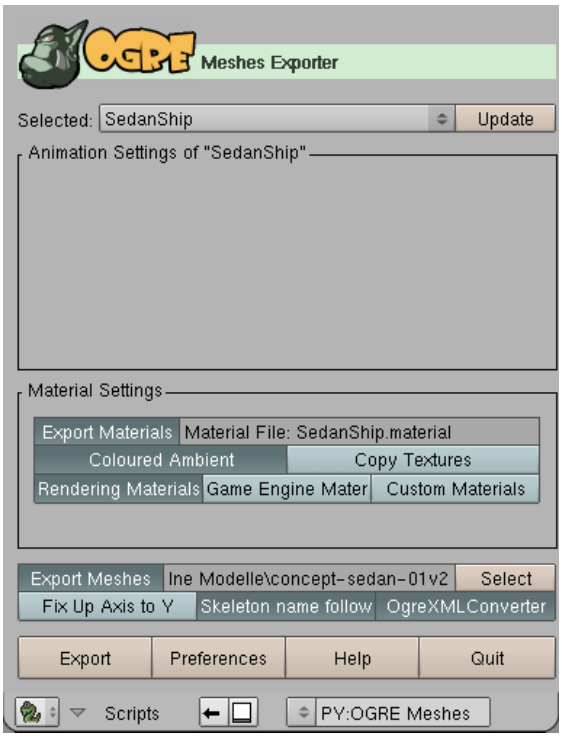


Figura 50: Captura de pantalla del exportador de Blender para OGRE 3D

4.1.4.2 Visual Studio 2008

Visual Studio es un entorno de desarrollo integrado para sistemas operativos Windows. Soporta varios lenguajes de programación como Visual C++, Visual C#, Visual J#, ASP .NET y Visual Basic .NET.

Visual Studio permita a los desarrolladores crear programas, aplicaciones web y servicios web en cualquier entorno que soporte la plataforma .NET y facilita la tarea de depuración.

Figura 51: Logotipo de Visual Studio

4.2 Desarrollo del programa

En esta sección se va a explicar el desarrollo del videojuego como programa. En concreto se va a explicar los subsistemas de gráficos, físicas, el núcleo del juego y el procesado de la entrada.

Se detallan los elementos más relevantes para el desarrollo del videojuego tanto del motor gráfico como del motor de físicas integrados y se muestran los diagramas de clases y secuencia del núcleo.

A continuación se muestra una captura del videojuego finalizado donde se pueden ver a los jugadores de ambos equipos coordinándose para ganar el partido.

Figura 52: Captura de pantalla del videojuego desarrollado

4.2.1 Gráficos

Para el desarrollo del subsistema de gráficos se ha decidido utilizar el motor gráfico Ogre3D. A continuación se hace una descripción de los principales componentes de Ogre3D para poder explicar cómo se ha hecho la integración con el videojuego desarrollado.

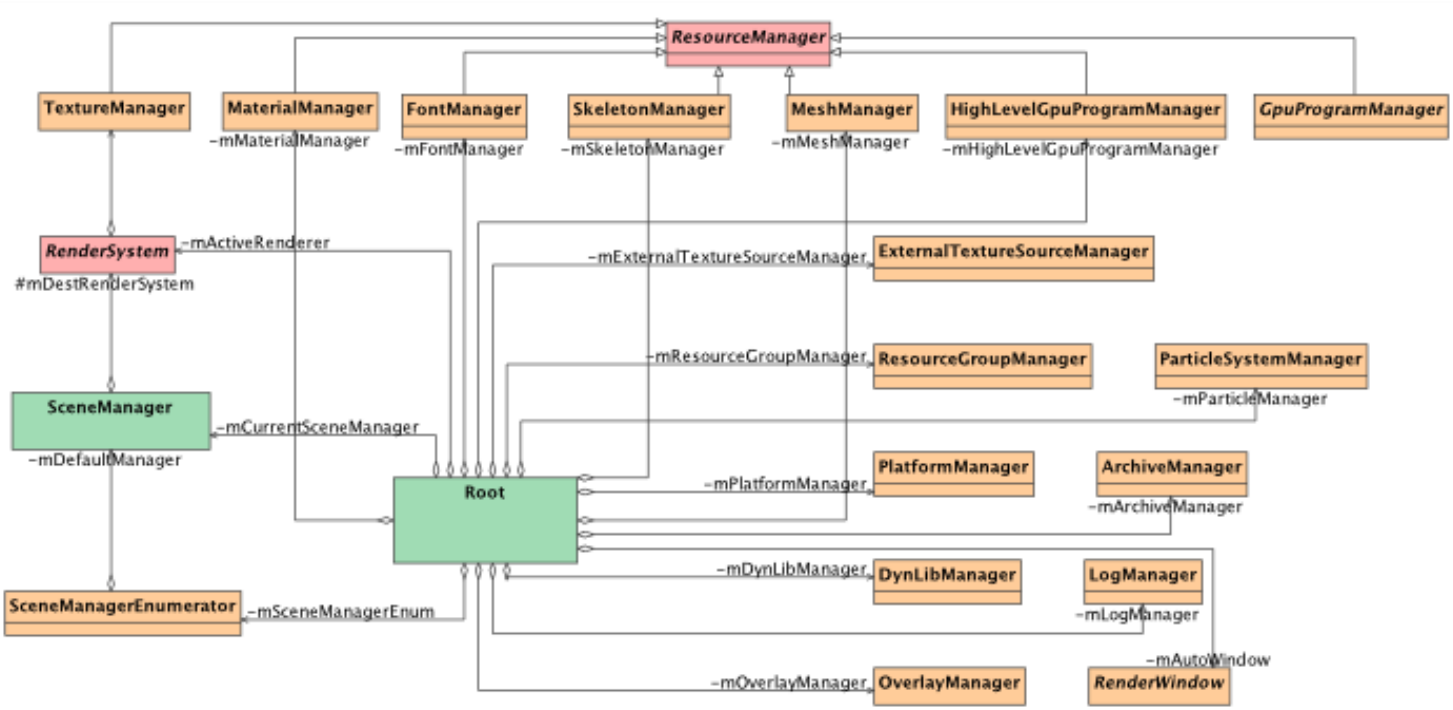


Figura 53: UML - Diagrama de clases de OGRE3 D

4.2.1.1 Objeto Root

El objeto Root es el punto de entrada del sistema Ogre. Es el primero en ser creado y el último en ser destruido. Detecta los diferentes sistemas de pintado disponibles y muestra una ventana al usuario donde le permite modificar la resolución, la profundidad de color, opciones de pantalla completa entre otras opciones. También permite obtener punteros a otros objetos esenciales como son el SceneManager y el RenderSystem.

4.2.1.1.1 Gestión de la escena

Las clases relativas a la gestión de la escena tienen como objetivo definir los contenidos de la escena, estructurarlos y definir como van han ser visualizados.

SceneManager

Después del Root el objeto SceneManager es probablemente el más importante y con el que la aplicación tiene la mayor parte de interacción. El SceneManager se encarga de gestionar todo el contenido de la escena que se tenga que pintar por pantalla. Es el responsable de organizar los contenidos con la técnica más adecuada para crear y gestionar las cámaras, objetos móviles, luces, materiales, etc.

Para crear cualquier objeto, la aplicación tiene que pedirlo directamente al SceneManager. Además, no es necesario mantener una lista de todos los objetos creados porque proporciona métodos para obtener un objeto a partir de su nombre.

El SceneManager también se encarga de enviar todo el contenido de la escena al RenderSystem cada vez que se tiene que pintar la escena. Dado que esto se realiza automáticamente, la mayoría de interacciones entre la aplicación y el SceneManager son las utilizadas para configurar y modificar la escena.

Diferentes tipos de escenas requieren algoritmos diferentes para decidir qué objetos se tienen que enviar a pintar de forma eficiente. El SceneManager por defecto no aplica demasiadas optimizaciones, pero existen especializaciones que en función de determinadas propiedades de la escena mejoran el rendimiento. Un ejemplo es el BSPSceneManager que optimiza las escenas interiores aplicando la técnica de partición binaria de espacios.

SceneNode

Para que un objeto forme parte de la escena se debe adjuntar a un SceneNode. Los SceneNode se estructuran de forma jerárquica entre ellos de manera que el movimiento de un SceneNode padre afecta también a la posición de sus hijos.

Por ejemplo en el caso de un vehículo se crearía un SceneNode para la carrocería y un SceneNode para cada una de las ruedas con su correspondiente posición. Si se aplica una rotación a un SceneNode de una rueda solo se moverá dicha rueda, pero si por el contrario se aplica una rotación al SceneNode de la carrocería entonces también se mueven todas las ruedas.

Entity

Las Entity están basadas en mallas de triángulos representadas por un objeto Mesh. En una misma escena puede haber múltiples Entity basadas en el mismo Mesh ya que es habitual utilizar copias de un mismo objeto. Por ejemplo, se puede utilizar el mismo modelo de árbol varias veces dentro de un bosque. Para que una Entity forme parte de la escena es necesario adjuntara a un SceneNode.

Light

Las luces se añaden a la escena como cualquier otro objeto. El objeto Light representa una luz que puede ser de diferentes tipos y que guarda información sobre sus propiedades (posición, color, intensidad, etc). Para determinar la posición de una luz hay dos posibilidades, definiendo sus coordenadas o adjuntándola a un SceneNode.

Los tipos disponibles de luces en Ogre son:

- **Point.** Son luces que emiten en todas direcciones
- **Spotlight.** Funcionan como la luz de una linterna. Tienen una posición de origen y una dirección donde se emite la luz.
- **Directional.** Simulan fuentes de luz lejanas que inciden sobre todos los objetos de la escena con un mismo ángulo, como es el caso de la luz solar.

Material

Los materiales especifican como se pintan los objetos en la pantalla. Definen las propiedades básicas de las superficies, como reflejan la luz o si brilla por sí mismo. También se especifica el número de texturas que se tienen que utilizar, como se tienen que utilizar y las imágenes que corresponden a cada una de ellas. Toda la

información relativa a la apariencia de un objeto exceptuando su geometría queda definida en el material.

La manera habitual de cargar un material es abriendo un fichero en el formato específico para Ogre (.material). Se trata de un lenguaje de scripting sencillo pero potente. De esta manera se separan todas las propiedades del material del código del programa, facilitando así su edición.

Ogre permite tener objetos con la misma malla de triángulos pero con diferentes materiales.

4.2.1.1.2 Gestión de recursos

El proceso de pintado requiere un gran número de recursos: imágenes, mallas de triángulos, fuentes, etc. El propósito de las clases relativas a la gestión de recursos es permitir la carga de estos recursos, su reutilización y su libramiento de memoria final.

ResourceGroupManager

El ResourceGroupManager es la clase que actúa como punto común para cargar recursos reusables como texturas y mallas de triángulos. Permite definir grupos de recursos de manera que estos puedan ser cargados o liberados cuando sea necesario.

El ResourceGroupManager tiene un conjunto de ResourceManagers que controlan cada uno de ellos los diferentes recursos. Se entiende por recurso todos los datos que tienen que ser cargados para proveer a Ogre con toda la información necesaria. Por ejemplo, el TextureManager se encarga de las texturas y el MeshManager de las mallas de triángulos.

ResourceManager

Los ResourceManagers posibilitan que los recursos sean cargados más de una vez y se comparten entre todos los componentes del sistema gráfico. Además, también gestionan la memoria requerida para la carga de estos recursos. Finalmente, se

pueden configurar al mismo tiempo diferentes carpetas donde buscar los recursos, incluyendo la búsqueda en archivos comprimidos.

Típicamente la aplicación no necesita acceder a los diferentes ResourceManagers sino que estos ya se llaman desde las diferentes partes de Ogre.

Mesh

Estos objetos representan las mallas de triángulos o modelos 3D que actúan como objetos móviles dentro de la escena. Los Mesh son un tipo de recursos y son gestionados por el MeshManager. Normalmente se cargan de ficheros con el formato específico de Ogre (.mesh). Para crear este tipo de ficheros se utilizan exportadores para los modeladores 3D.

Por último, los Mesh también pueden incluir animaciones a partir de un esqueleto o hechas por transformación.

Texture

Una textura es una imagen que puede ser aplicada en una superficie o en una malla de triángulos. En Ogre, las texturas se representan con el objeto Texture.

Los Texture son creados a partir del TextureManager. En muchos casos son creados a partir de una imagen cargada por el gestor de recursos de Ogre aunque también se pueden crear manualmente.

4.2.1.1.3 Pintado

Las clases encargadas del pintado se encargan de los aspectos a más bajo nivel relacionados con el pintado por pantalla.

RenderSystem

El RenderSystem es una clase abstracta que define una interfaz con el sistema de pintado. Se encarga de enviar todas las operaciones a la interfaz de pitado y de aplicar las diferentes opciones de configuración. Se trata de una clase abstracta ya

que su implementación depende del sistema de pintado y existen subclases para cada uno de los sistemas soportados. Por ejemplo, la clase D3DRenderSystem en el caso de DirectX.

OverlayManager

El objeto OverlayManager permite gestionar la creación de Overlays y sus contenedores. Un Overlay es un elemento 2D o 3D que se pinta por encima de la escena para crear efectos como menús, paneles de estados o la interfaz visual del videojuego.

Camera

Las cámaras son instancias de la clase Camera. Toda escena debe tener al menos una cámara que define el punto de vista desde donde se va a realizar el pintado de la escena. Se pueden definir varias cámaras aunque solo se puede pintar lo que se visualiza desde una a la vez por viewport.

Viewport

Un viewport define la región de la superficie de pintado donde se va a pintar la escena vista desde una cámara.

4.2.2 Físicas

Para simular las físicas y detectar las colisiones en el videojuego se va a utilizar el motor de físicas Bullet. Como se busca simplicidad en el desarrollo del videojuego y gracias a la modularidad que ofrece el motor de físicas Bullet, sólo se utilizarán los componentes de detección de colisiones y dinámicas de cuerpos rígidos.

Detección de colisiones

El módulo de detección de colisiones provee de algoritmos y estructuras para acelerar las consultas del punto más cercano así como test de rayos y barrido convexo. Las estructuras de datos principales son:

- btCollisionObject es el objeto que tiene la información de las transformaciones del objeto (posición y orientación) y la forma de colisión.
- btCollisionShape describe la forma de colisión de un objeto, como por ejemplo una caja, una esfera o una malla de triángulos. Una misma forma de colisión puede ser compartida por varios objetos.
- btGhostObject es un objeto especial útil para hacer consultas de localización de una colisión.
- btCollisionWorld almacena todos los btCollisionObjects y provee de una interfaz para realizar consultas.

Bullet soporta una gran variedad de formas de colisión, las más comunes son las siguientes:

- btBoxShape. Caja definida como la mitad de la altura de sus lados.
- btSphereShape. Esfera definida por su radio.
- btCapsuleShape. Capsula alrededor del eje Y.
- btCylinderShape. Cilindro alrededor del eje Y.
- btConeShape. Cono alrededor del eje Y.
- btConvexHullShape. Envoltorio convexo para una malla de triángulos.

- btStaticPlaneShape. Plano estático infinito que divide el espacio en dos mitades.

Dinámicas de cuerpos rígidos

El módulo de dinámicas de cuerpos rígidos está implementado por encima del módulo de detección de colisiones. Añade fuerzas, masas, inercias, velocidades y restricciones.

Existen tres tipos distintos de cuerpos rígidos en Bullet:

- Dinámicos. Tiene masa y en cada ejecución del motor se actualiza su transformación.
- Estáticos. No tienen masa y no se pueden mover pero sí colisionar.
- Cinemáticos. No tienen masa y pueden ser movidos por el usuario pero no tienen interacción con los objetos dinámicos.

Todos ellos tienen que ser añadidos al llamado mundo dinámico de Bullet con una forma de colisión, esta forma puede es utilizada para calcular la distribución de masas. Las transformaciones (posición y orientación) que provee Bullet una vez se ha realizado un paso de la simulación son respecto al centro de masas de los objetos.

Para facilitar la tarea de sincronizar la posición y orientación de los objetos entre el mundo dinámico y su representación gráfica en el mundo 3D, Bullet ofrece el llamado btMotionState que hace de unión entre los dos mundos. Cuando se crea un btRigidBody, también se genera un btMotionState asociado a él. En el primer fotograma, el motor de físicas obtiene la posición y orientación de los objetos del mundo 3D a través del btMotionState. Cuando se hace una simulación de las físicas, Bullet actualiza el btMotionState de cada objeto para poder así sincronizar las transformaciones con el mundo 3D.

Integración

El proceso de integración del motor de físicas Bullet consiste en crear un mundo dinámico instanciando la clase btDiscreteDynamicsWorld y añadiendo para cada

objeto que se quiere mover un btRigidBody. Para cada btRigidBody se tiene que definir su masa, su forma de colisión (caja, esfera, envoltorio convexo, etc) y las propiedades del material como la fricción o la elasticidad en las colisiones.

Para simular las físicas es necesario ejecutar un paso de simulación en cada fotograma del videojuego mediante la función stepSimulation. Entonces btDiscreteDynamicsWorld realiza una interpolación de los movimientos respecto a la simulación anterior y realiza la detección de colisiones y las dinámicas. Una vez se ha realizado la simulación, Bullet actualiza el btMotionState de cada objeto con las transformaciones generadas.

4.2.3 Núcleo del juego

El núcleo del juego es el encargado de gestionar la lógica específica del videojuego en cuestión. Al tratarse de un videojuego de fútbol, encontramos entidades como los jugadores, la pelota, el campo o las porterías así como entidades no físicas como el partido o los equipos además de clases que permiten el control de los jugadores, o la gestión del videojuego.

A continuación se muestra el diagrama de clases del núcleo del videojuego así como una descripción de cada clase.

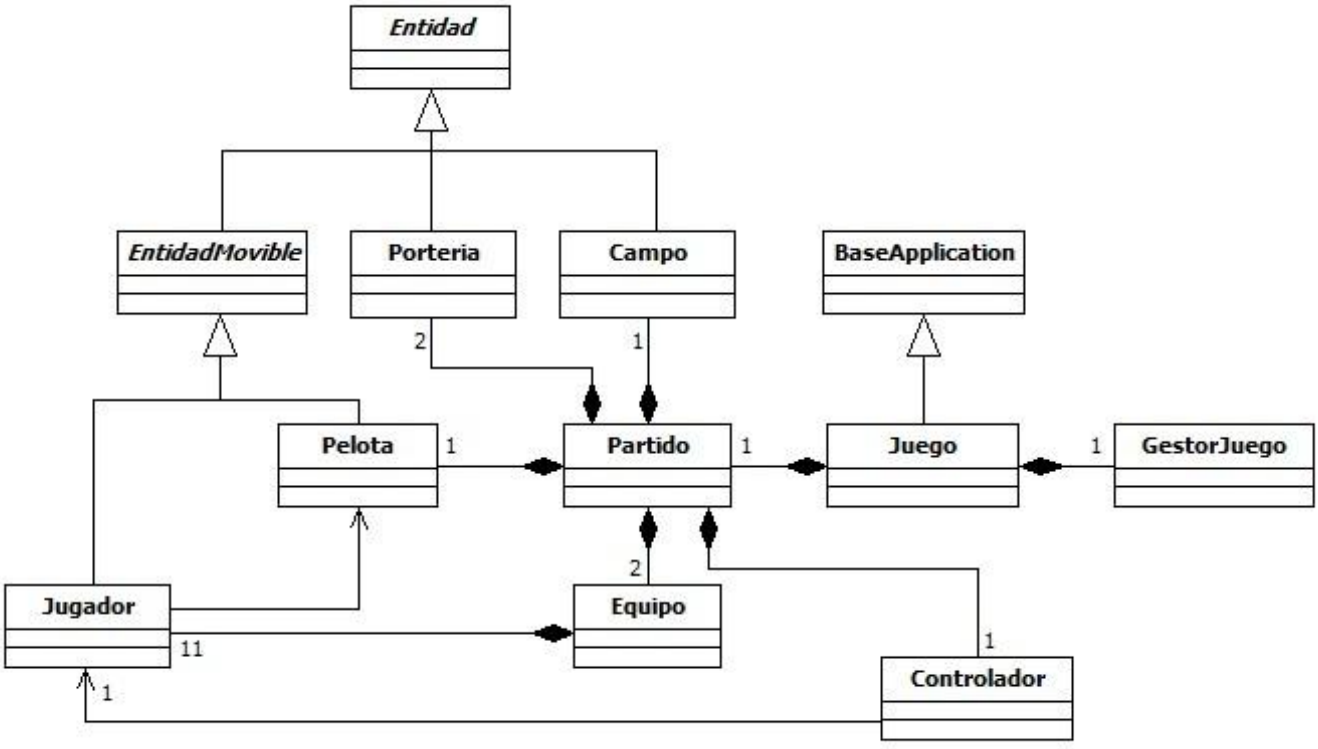


Figura 54: UML - Diagrama de clases del núcleo del juego

BaseApplication

Es una clase abstracta que facilita la creación de aplicaciones con Ogre3D. Implementa varias interfaces que permiten la recepción de eventos como por ejemplo el inicio de pintado de un fotograma, el redimensionamiento de la pantalla o eventos generados por el teclado y el ratón.

También es la encargada de mantener objetos esenciales de Ogre3D como son el Root, el SceneManager, el viewport o la cámara, crear la escena, inicializar y configurar el sistema de pintado, cargar los recursos necesarios, etc.

Toda clase que herede de BaseApplication será el punto de entrada de la aplicación y la encargada de actualizar el estado del videojuego mediante la recepción de eventos como "frameRenderingQueued" que se produce una vez por fotograma permitiendo implementar en su interior el bucle típico de un videojuego de captura de eventos, actualización de la lógica y pintado

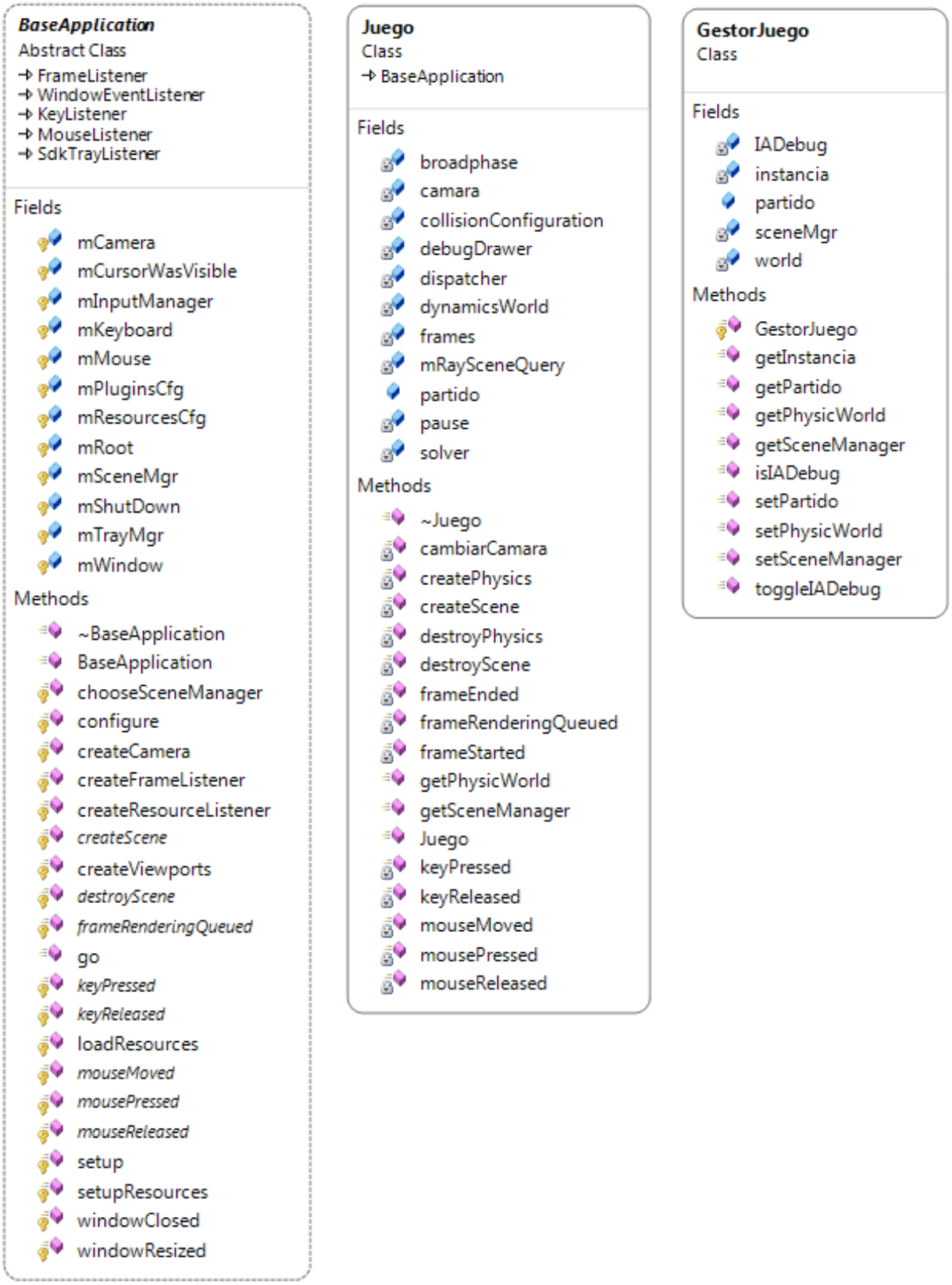
Juego

La clase Juego hereda de BaseApplication y por lo tanto es el punto de entrada de la aplicación y la encargada de actualizar el estado del videojuego.

Es la clase encargada de crear y actualizar los objetos específicos del videojuego implementado como son el motor de físicas o el partido de fútbol. También se encarga de tratar los eventos generados por el teclado y el ratón y aplicar las acciones pertinentes dentro del contexto del videojuego.

GestorJuego

La clase GestorJuego es una clase Singleton cuyo propósito es mantener referencias al SceneManager de Ogre3D para poder añadir SceneNodes desde cualquier punto del videojuego, mantener también una referencia al mundo de físicas para poder añadir formas de colisión durante la creación de las entidades físicas y por último mantener una referencia al partido que contiene las entidades que participan en él como son los jugadores o la pelota.


BaseApplication
Abstract Class
FrameListener
WindowEventListener
KeyListener
MouseListener
SdkTrayListener
Fields
mCamera
mCursorWasVisible
mInputManager
mKeyboard
mMouse
mPluginsCfg
mResourcesCfg
mRoot
mSceneMgr
mShutDown
mTrayMgr
mWindow
Methods
~BaseApplication
BaseApplication
chooseSceneManager
configure
createCamera
createFrameListener
createResourceListener
createScene
createViewports
destroyScene
frameRenderingQueued
go
keyPressed
keyReleased
loadResources
mouseMoved
mousePressed
mouseReleased
setup
setupResources
windowClosed
windowResized
Juego
Class
BaseApplication
Fields
broadphase
camara
collisionConfiguration
debugDrawer
dispatcher
dynamicsWorld
frames
mRaySceneQuery
partido
pause
solver
Methods
~Juego
cambiarCamara
createPhysics
createScene
destroyPhysics
destroyScene
frameEnded
frameRenderingQueued
frameStarted
getPhysicWorld
getSceneManager
Juego
keyPressed
keyReleased
mouseMoved
mousePressed
mouseReleased
GestorJuego
Class
Fields
IADebug
instancia
partido
sceneMgr
world
Methods
GestorJuego
getInstancia
getPartido
getPhysicWorld
getSceneManager
isIADebug
setPartido
setPhysicWorld
setSceneManager
toggleIADebug

A continuación se muestra el diagrama de secuencia de la inicialización del videojuego.

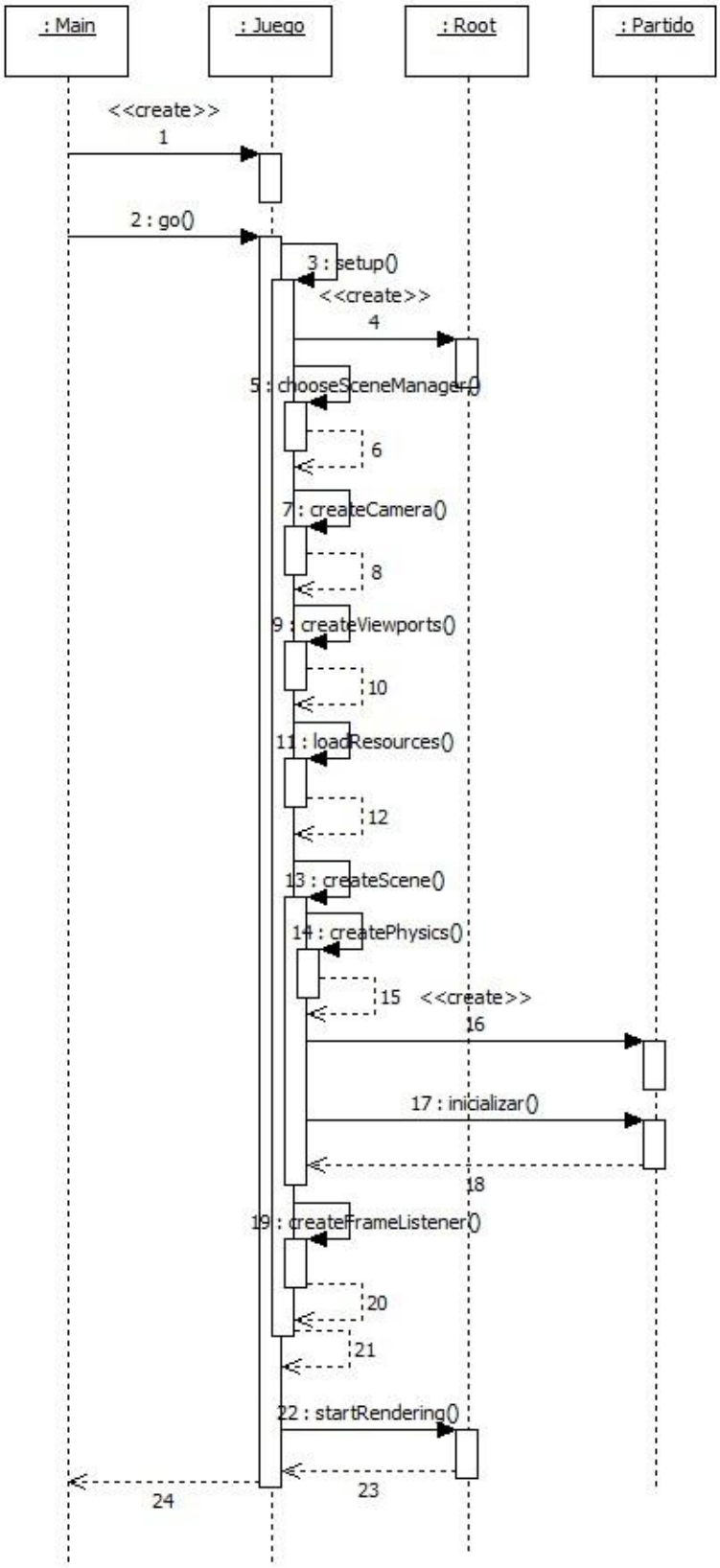


Figura 55: UML - Diagrama de secuencia de la inicialización del videojuego

A continuación se muestra el diagrama de secuencia de la actualización del videojuego en cada fotograma.

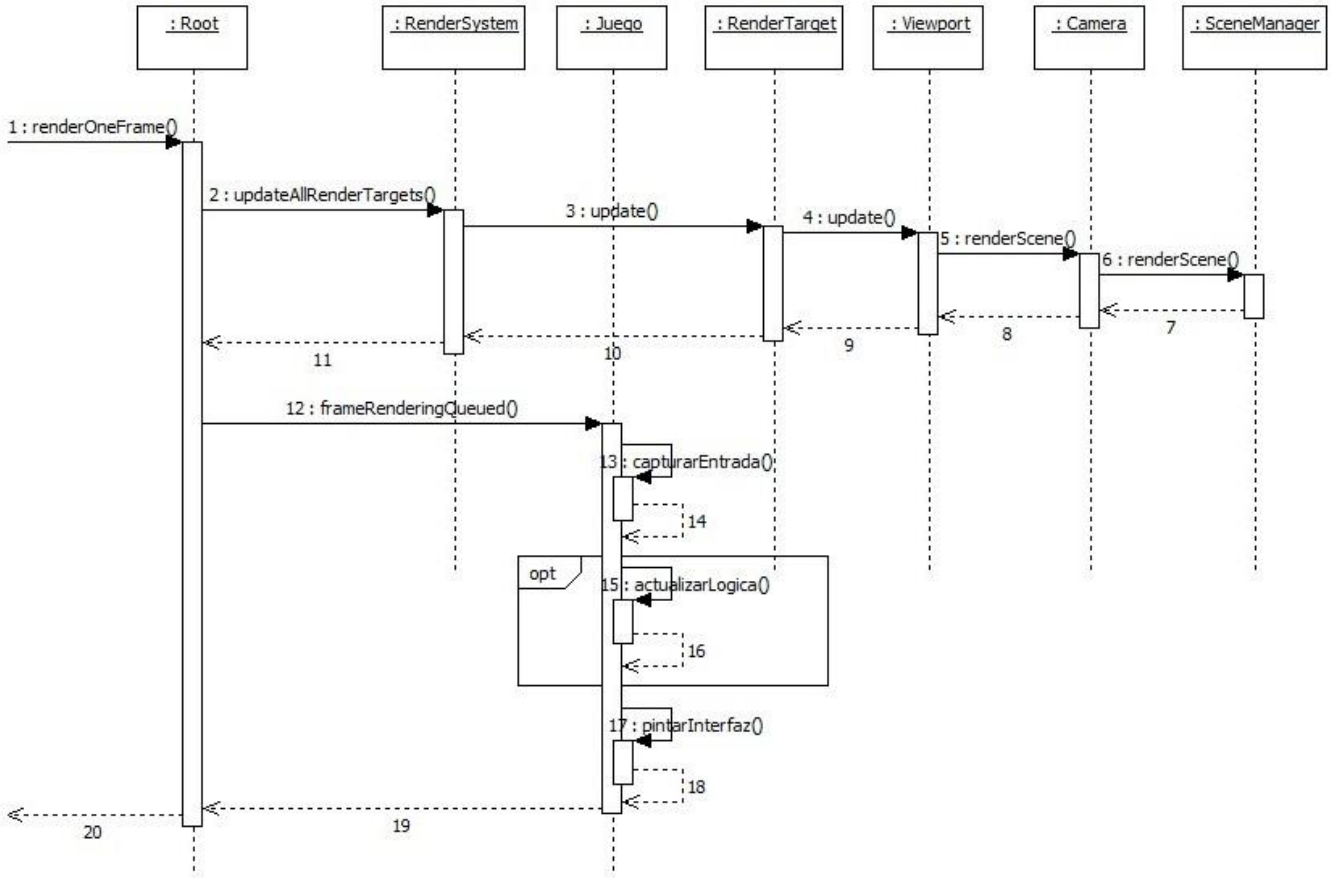


Figura 56: UML - Diagrama de secuencia de del bucle de actualización del videojuego

En cada fotograma, Ogre3D llama al método "renderOneFrame" del objeto Root que se encarga de comunicarle al RenderSystem que es necesario pintar por pantalla el estado actual de la escena vista desde una cámara determinada.

Mientras toda la información gráfica es enviada a la GPU, el objeto Root ejecuta el método "frameRenderingQueued" del Juego para terminar el bucle típico de un videojuego que consiste en capturar los eventos producidos por el usuario, actualizar la lógica del partido como las físicas, la inteligencia artificial, etc., y por último pintar los elementos de la interfaz gráfica.

Entidad

La clase Entidad es una clase abstracta de la cual heredan todos los elementos del videojuego que tienen una representación gráfica y física.

Toda entidad está representada por una Entity de Ogre3D que está asociada a un SceneNode el cual permite mover a la entidad por la escena.

Para detectar las colisiones y generar las dinámicas, toda entidad está representada por un RigidBody de Bullet que tiene asociado un RigidBodyConstructionInfo que especifica las características físicas de la entidad, así como una CollisionShape que determina la forma de colisión que tendrá la entidad.

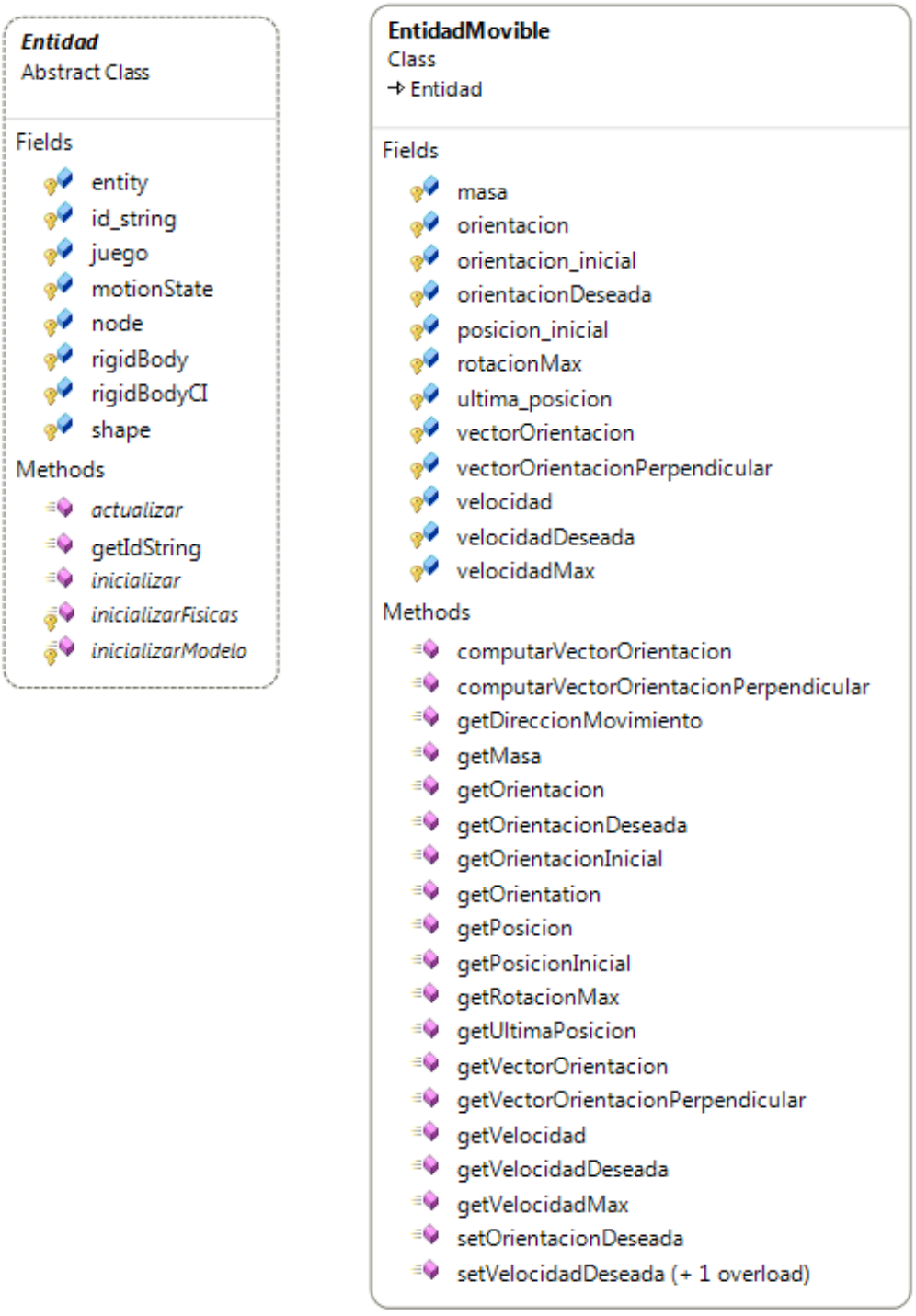
Por último, tiene un MotionState que permite sincronizar las transformaciones del mundo físico de Bullet con el mundo 3D de Ogre3D.

EntidadMovible

La clase EntidadMovible es una clase abstracta que hereda de Entidad y que provee a las clases que heredan de ella de atributos y métodos para consultar sus características cinemáticas.

Mantiene la posición y orientación iniciales útiles para el reiniciado de la entidad, su masa, orientación y velocidad actuales, así como la velocidad y rotación máximas para limitar su movimiento.

Por último, permite gestionar la velocidad y orientación deseadas útiles para el control de la entidad ya sea por el usuario o por la inteligencia artificial.


Entidad
Abstract Class
Fields
entity
id_string
juego
motionState
node
rigidBody
rigidBodyCI
shape
Methods
actualizar
getIdString
inicializar
inicializarFisicas
inicializarModelo
EntidadMovible
Class
→ Entidad
Fields
masa
orientacion
orientacion_inicial
orientacionDeseada
posicion_inicial
rotacionMax
ultima_posicion
vectorOrientacion
vectorOrientacionPerpendicular
velocidad
velocidadDeseada
velocidadMax
Methods
computarVectorOrientacion
computarVectorOrientacionPerpendicular
getDireccionMovimiento
getMasa
getOrientacion
getOrientacionDeseada
getOrientacionInicial
getOrientation
getPosicion
getPosicionInicial
getRotacionMax
getUltimaPosicion
getVectorOrientacion
getVectorOrientacionPerpendicular
getVelocidad
getVelocidadDeseada
getVelocidadMax
setOrientacionDeseada
setVelocidadDeseada (+ 1 overload)

Campo

La clase Campo es la representación lógica del campo de fútbol. Hereda de la clase Entidad porque tiene representación gráfica y física pero no se mueve por la escena.

Tiene un método útil para detectar si una entidad se ha salido fuera del área delimitada por las líneas del campo de fútbol.

Por último, tiene atributos y métodos para la gestión de la división del campo en regiones lógicas útiles para la toma de decisiones de la inteligencia artificial.

Porteria

La clase Porteria es la representación lógica de una portería de fútbol. Aunque hereda de Entidad la cual ofrece una representación física y gráfica de la misma, la clase portería gestiona la representación físicas y gráfica de cada uno de sus postes.

También provee de un método útil para comprobar si la pelota ha pasado entre sus postes para así poder detectar los posibles goles.

Pelota

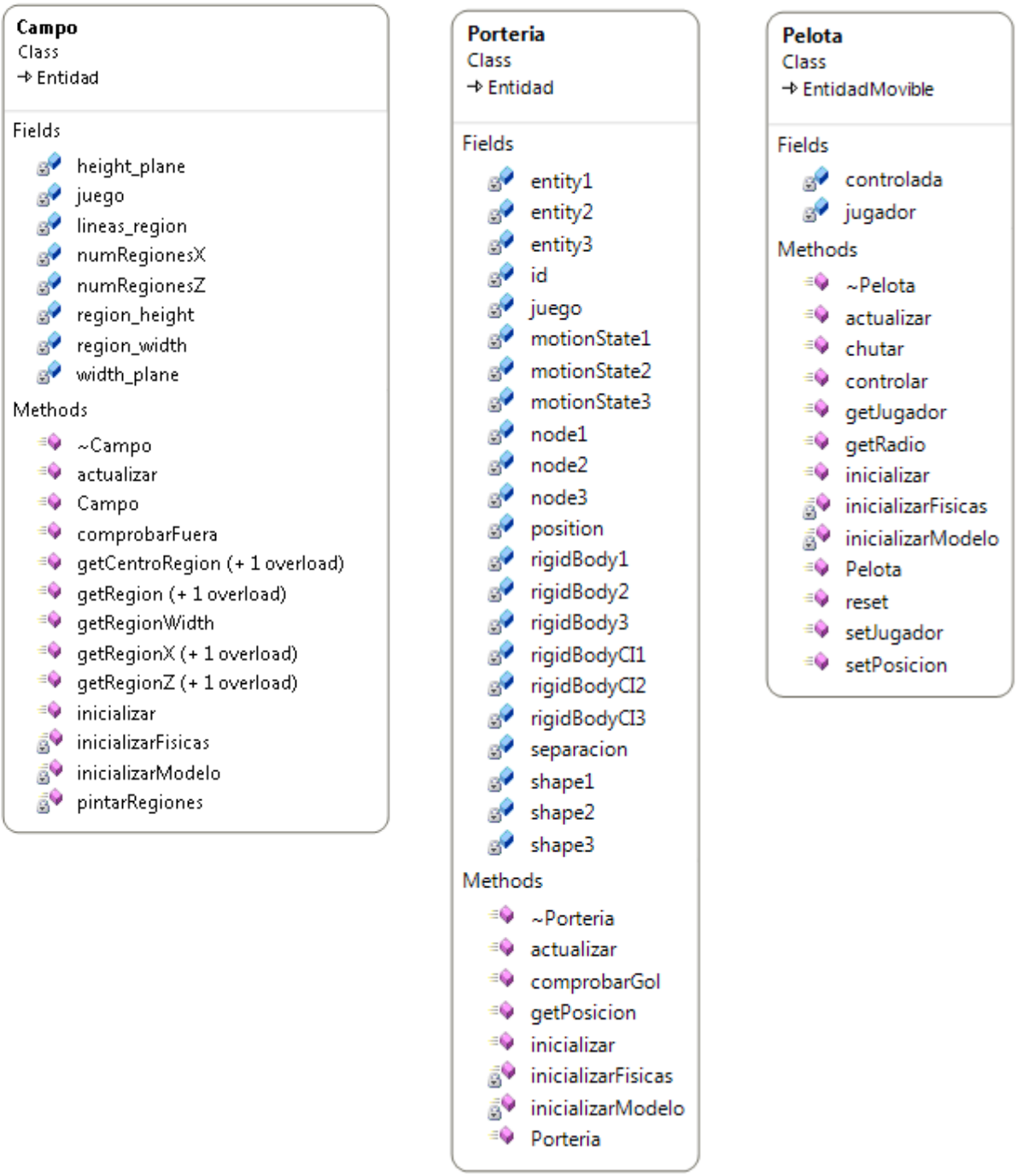
La clase Pelota es la representación lógica de la pelota de fútbol. Hereda de la clase EntidadMovible ya que es una entidad que se va a mover continuamente por la escena.

La conducción de la pelota por parte de un jugador es gestionada por la propia pelota.

Mediante el método “setJugador” se le indica a la pelota qué jugador es el que tiene el control. Si la pelota tiene asignado a un jugador, cada vez que se llama al método “actualizar” la pelota se aplica a sí misma una fuerza orientada hacia la posición de conducción del jugador. La posición de conducción de un jugador está un poco adelantada respecto a su posición actual y orientación.

Si la pelota no está controlada por un jugador entonces su movimiento se rige por la dinámica gestionada por el motor de físicas.

Por último, dispone del método “chutar” que permite aplicar una fuerza a la pelota y así liberarla del control del jugador.



Equipo

La clase Equipo no hereda ni de Entidad ni de EntidadMovible porque no tiene representación ni gráfica ni física. Es la encargada de gestionar a sus jugadores así como mantener una referencia a su portería y la del rival.

Provee de métodos útiles como "getJugadorMasCercano" que indica qué jugador del equipo es el más cercano a una posición. Si esa posición corresponde a la posición de la pelota entonces se puede saber qué jugador es el más cercano a la pelota.

Jugador

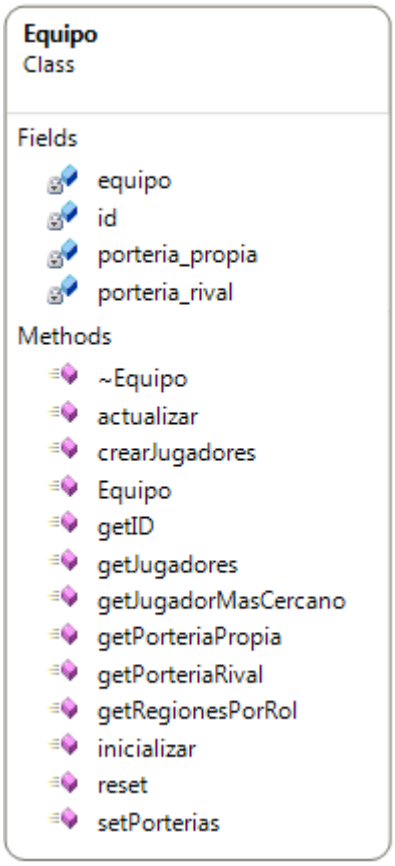
La clase Jugador es la representación lógica de los jugadores de fútbol y al igual que la pelota hereda de EntidadMovible.

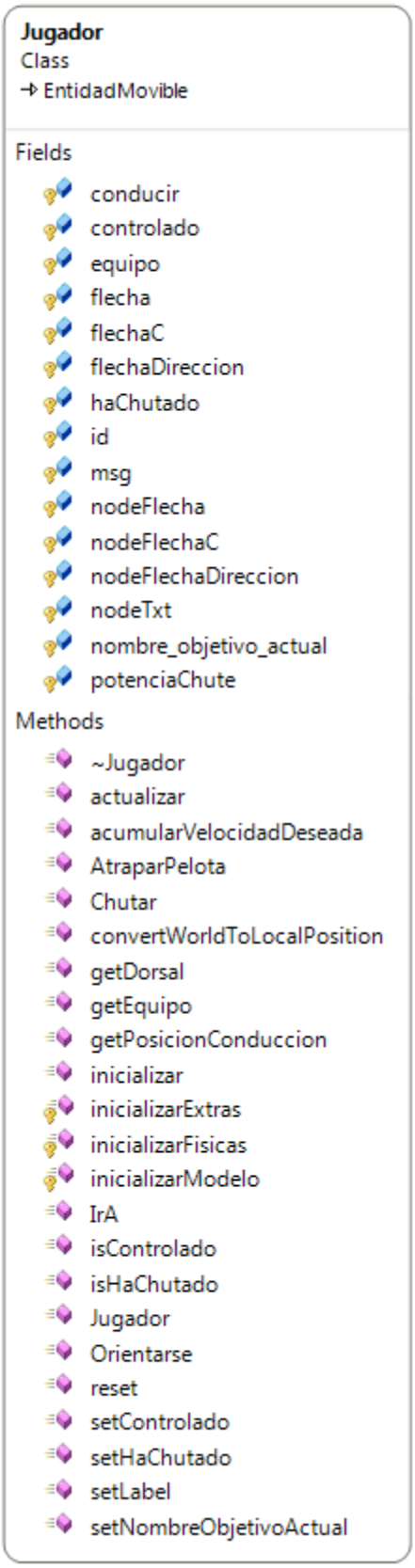
El control de movimiento del jugador ya sea por parte del usuario o por parte de la inteligencia artificial, se realiza mediante la indicación de la velocidad y orientación deseadas. De esta manera, cada vez que se ejecuta el método "actualizar", el jugador se mueve y se orienta para alcanzar la velocidad y orientación deseadas. Una vez se ha calculado la velocidad y orientación necesarias para cada fotograma, se le comunica al motor de físicas mediante fuerzas y se actualiza su MotionState.

Para facilitar la visualización su orientación actual se muestra un triángulo del mismo color en la base del jugador. Solucionando el inconveniente de no poder saber hacia dónde está mirando el jugador debido a su modelo 3D.

Así mismo, para indicar qué jugador está controlado por el usuario se muestra un triángulo de color amarillo encima del jugador.

Por último, se muestra un texto flotante encima de cada jugador que muestra el dorsal del jugador o información relativa a su inteligencia artificial.


Equipo
Class
Fields
equipo
id
porteria_propia
porteria_rival
Methods
~Equipo
actualizar
crearJugadores
Equipo
getID
getJugadores
getJugadorMasCercano
getPporteriaPropia
getPorteriaRival
getRegionesPorRol
inicializar
reset
setPorterias

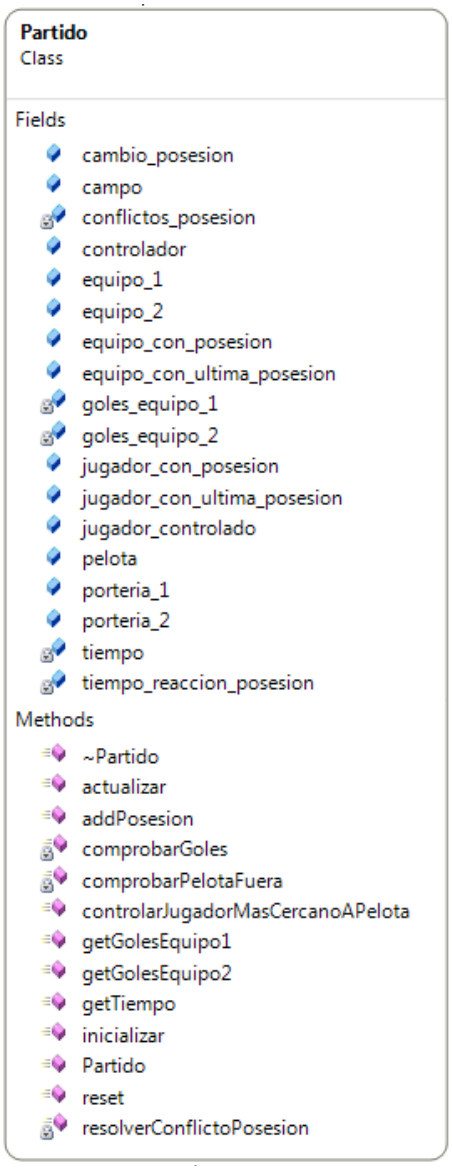

Jugador
Class
EntidadMovible
Fields
conducir
controlado
equipo
flecha
flechaC
flechaDireccion
haChutado
id
msg
nodeFlecha
nodeFlechaC
nodeFlechaDireccion
nodeTxt
nombre_objetivo_actual
potenciaChute
Methods
~Jugador
actualizar
acumularVelocidadDeseada
AtraparPelota
Chutar
convertWorldToLocalPosition
getDorsal
getEquipo
getPosicionConduccion
inicializar
inicializarExtras
inicializarFisicas
inicializarModelo
IrA
isControlado
isHaChutado
Jugador
Orientarse
reset
setControlado
setHaChutado
setLabel
setNombreObjetivoActual

Partido

La clase Partido es la encargada de mantener el estado del partido así como mantener referencias a todas las entidades que participan en él.

Es la encargada de reiniciar las entidades en el caso de producirse un gol así como de resolver conflictos de posesión de la pelota entre varios jugadores.

Cuando un jugador se encuentra lo suficientemente cerca de la pelota se registra como posible poseedor. Cada vez que se ejecuta el método “actualizar” se comprueba si hay más de un jugador que quiere controlar la pelota y en ese caso se produce un conflicto porque solo un jugador puede controlar la pelota a la vez. La resolución de este conflicto se realiza mediante la selección aleatoria de uno de los candidatos.



A continuación se muestra el diagrama de secuencia de la creación de un partido y todas sus entidades.

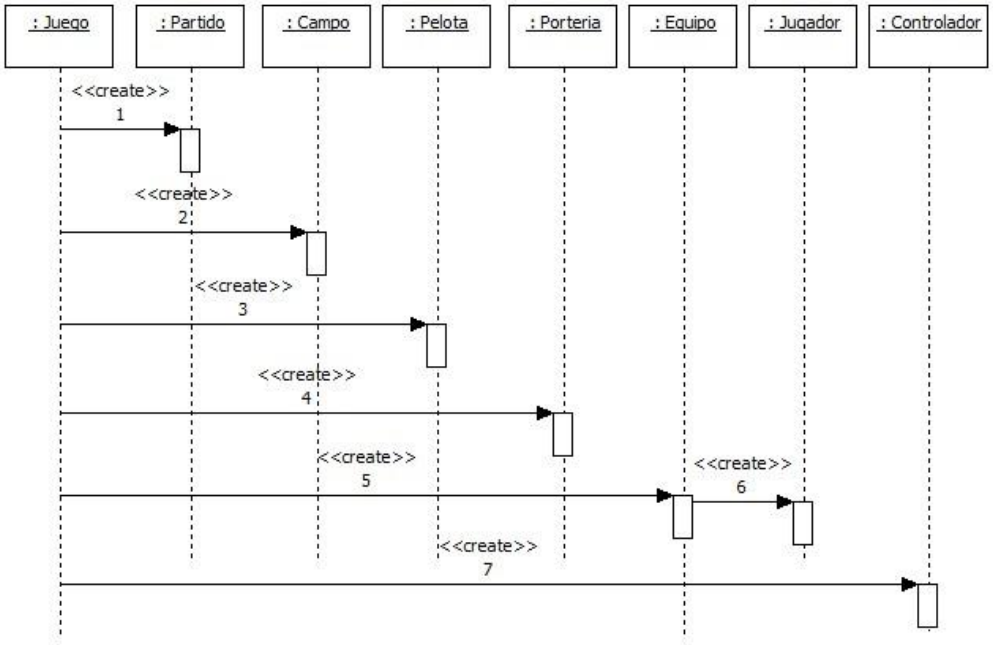


Figura 57: UML - Diagrama de secuencia de la creación de un partido

A continuación se muestra el diagrama de secuencia de la inicialización de un partido y de todas sus entidades. Durante la inicialización se cargan los modelos y se inicializan las físicas.

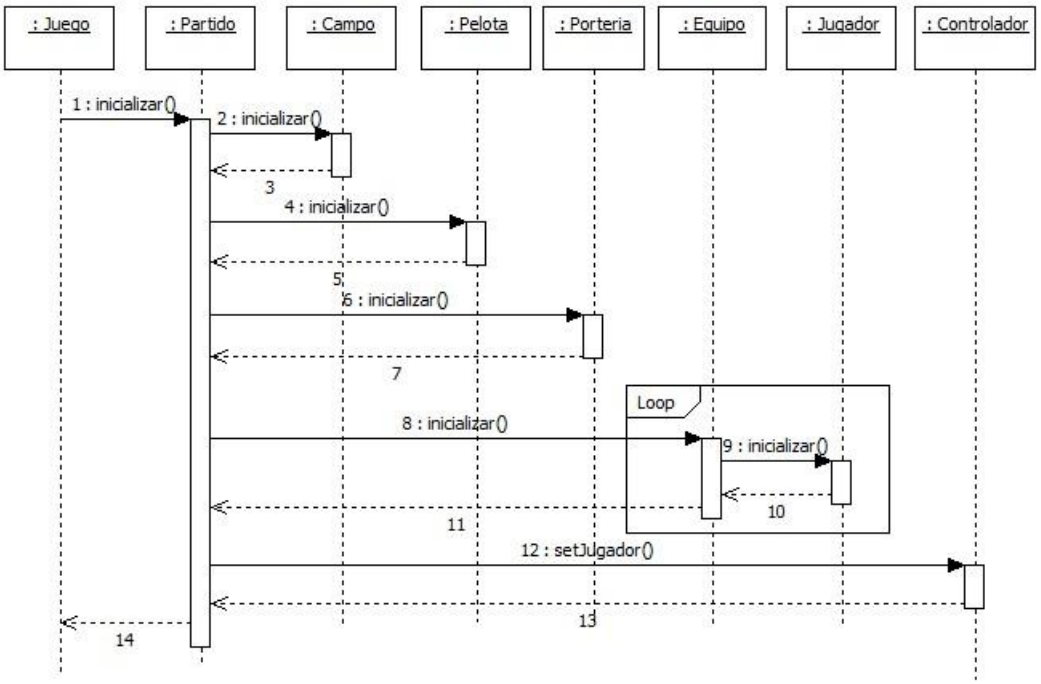


Figura 58: UML - Diagrama de secuencia de la inicialización de un partido

A continuación se muestra el diagrama de secuencia de la actualización del partido y todas las entidades que participan en él.

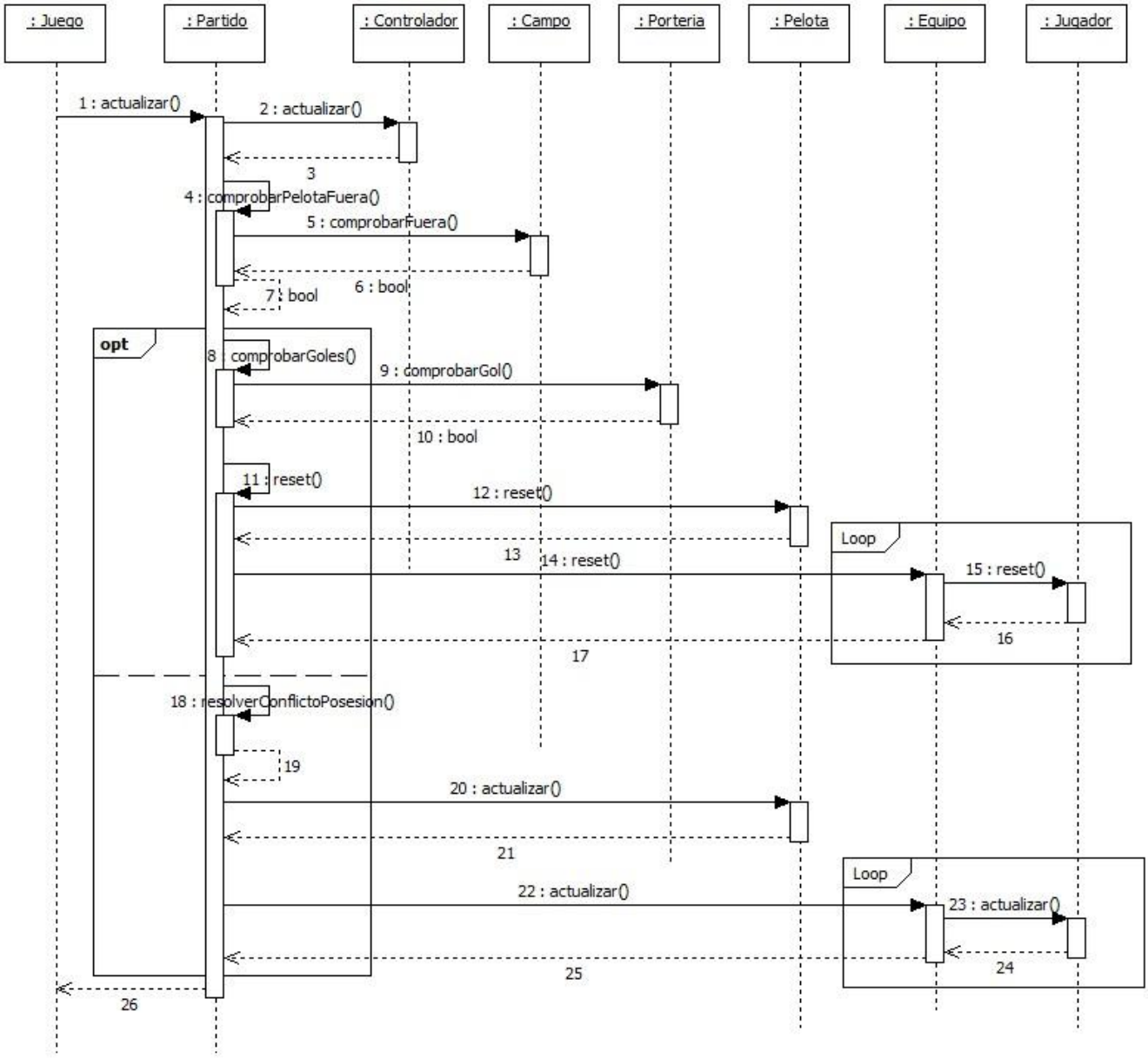


Figura 59: UML - Diagrama de secuencia de la actualización de un partido

En cada fotograma se actualiza el estado del partido mediante la llamada al método "actualizar" de la clase Partido. Lo primero que hace es procesar los eventos generados por el usuario llamando al método de "actualizar" de la clase Controlador.

Para comprobar si la pelota se ha salido del terreno de juego se le pide a la clase Campo dicha comprobación. Si la pelota está fuera del área jugable entonces se comprueba si se ha producido algún gol en alguna de las porterías. Si se ha producido un gol entonces se incrementa el número de goles al equipo que corresponda. Para simplificar el videojuego se ha decidido que si la pelota se sale del terreno de juego se reinicia el partido y por lo tanto todas las entidades que participan en él.

Si la pelota no se ha salido del terreno de juego entonces lo primero que se hace es resolver el conflicto de posesión de la pelota en el caso de que exista una disputa entre varios jugadores. Acto seguido se actualizan el resto de entidades que participan en el partido.

4.2.4 Procesamiento de entrada

La captura de los eventos producidos por el usuario se realiza mediante el uso de Object-oriented Input System Library (OIS).

La clase Juego implementa las interfaces KeyListener y MouseListener que permiten recibir eventos del teclado y del ratón respectivamente y procesarlos para realizar las acciones pertinentes dentro del contexto del videojuego.

A continuación se detallan las acciones que puede realizar el usuario del videojuego.

Teclado

- **ESC**: Se cierra el videojuego.
- **I**: se habilita/deshabilita la visualización de información de la inteligencia artificial.
- **R**: se reinicia el partido.
- **P**: se pausa el partido.
- **A**: se habilita/deshabilita el control de los jugadores por parte del usuario.
- **C**: se cambia la cámara.
- **SPACE**: se controla el jugador más cercano a la pelota.

Ratón

- **Clic izquierdo**: se indica la posición de movimiento del jugador controlado.
- **Clic derecho**: se indica la posición hacia dónde quiere chutar el usuario.

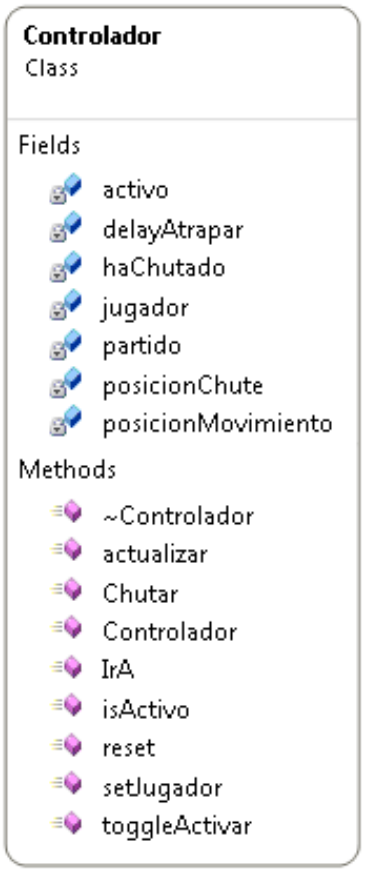
Para calcular dichas posiciones se traza un rayo desde la cámara a la escena y se calcula su intersección con el campo de fútbol. Una vez calculadas estas posiciones se notifica a la clase Controlador el cual realiza las acciones pertinentes para controlar a los jugadores.

Controlador

La clase Controlador permite facilitar el control de un jugador por parte del usuario. En concreto recibe las decisiones tomadas por el usuario en forma de posiciones como hacia donde debe ir el jugador y hacia donde debe chutar la pelota.

Al ejecutar el método "actualizar" se decide en función del contexto hacia donde debe mirar el jugador o si puede chutar la pelota. Es el encargado de detectar si el usuario quiere realizar un pase a otro jugador o quiere chutar a la portería.

Para lograr la coordinación entre el jugador controlado por el usuario y el resto de jugadores controlados por la inteligencia artificial, la clase Controlador modifica el conocimiento compartido entre todos los jugadores para notificar de las acciones realizadas por el usuario. Por lo tanto la clase Controlador hace también de nexo entre el usuario y el motor de inteligencia artificial.



4.2.5 Patrones de diseño

Modelo Vista Controlador (MVC)

El patrón Modelo Vista Controlador es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario y la lógica en tres componentes distintos.

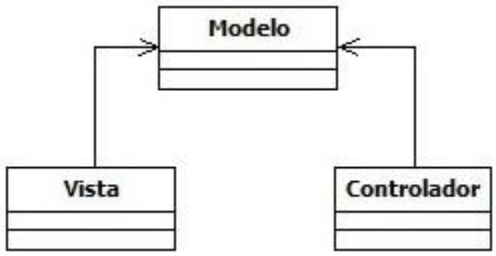


Figura 60: Patrón de diseño "Modelo Vista Controlador"

En el videojuego el modelo está representado por el núcleo del juego ya que es el encargado de mantener la información de las entidades que participan en él. La vista está gestionada por el motor gráfico Ogre3D que es el encargado de mostrar el estado del videojuego al usuario, y el controlador está representado por la clase Controlador que recibe los eventos generados por el usuario y modifica el modelo.

4.3 Desarrollo de contenidos

4.3.1 Ilustración, modelado y animación.

Campo

Para la representación del campo se ha obtenido una ilustración de un campo de fútbol con césped y sus principales líneas pintadas y se ha aplicado como textura en una de las caras de un hexaedro con una superficie igual al de la ilustración pero con una altura muy pequeña.

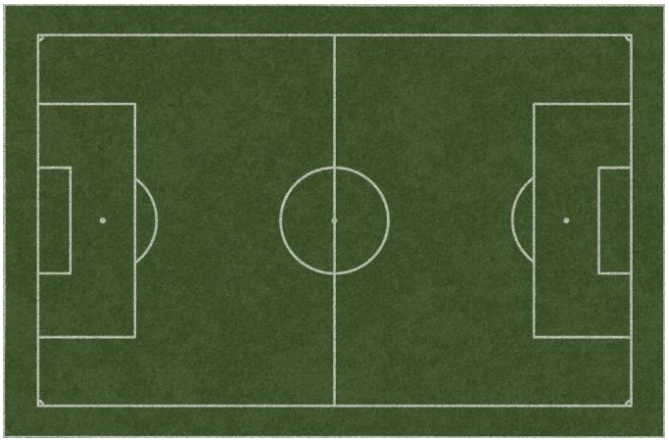


Figura 61: Captura de pantalla del campo de fútbol

Portería

Para la representación de las porterías, se ha creado con Blender el modelo 3D de un cilindro de color blanco. Éste cilindro se ha reaprovechado para cada uno de los tres postes de las porterías. En concreto se han creado dos postes verticales y un poste horizontal que se ha escalado para unir ambos postes verticales.

Figura 62: Captura de pantalla de una portería

Pelota

La pelota se representa mediante una esfera de color amarillo. Se ha optado por utilizar el color amarillo para poder diferenciarla fácilmente del resto de elementos de la escena.

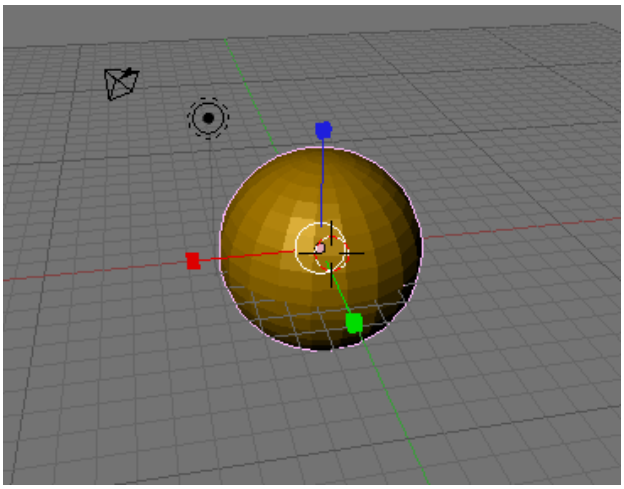


Figura 63: Captura de pantalla de la pelota en Blender

Jugadores

Todos los jugadores se representan mediante cilindros. Para diferenciar los jugadores de ambos equipos se ha optado por crear dos cilindros con materiales de colores diferentes, los de un equipo de color rojo y los del otro de color azul.

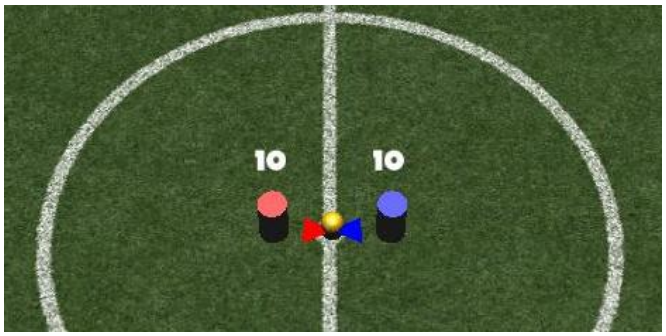


Figura 64: Captura de pantalla de los jugadores

Fondo

Para ambientar la escena se ha puesto una fotografía aérea del cielo diurno que una vez se superpone el campo de fútbol da la sensación de estar flotando y permite realzarlo.

Figura 65: Captura de pantalla del fondo de la escena

4.3.2 Interfaz

Tiempo y Marcador

Para representar el tiempo transcurrido desde la inicialización del partido y la contabilización de los goles marcados por cada equipo, se han creado dos overlays de Ogre3D que se han situado en la parte superior de la pantalla.

Figura 66: Captura de pantalla de la interfaz de usuario

5 Desarrollo de la inteligencia artificial especifica

5.1 Análisis

El fútbol es una pugna entre dos equipos compuestos por once jugadores cada uno. En esa pugna hay una pelota por medio y unas reglas. Dentro del terreno de juego, los futbolistas de cada equipo tienen que conseguir meter más goles que los del otro equipo, estableciéndose para ello unas relaciones de oposición y de cooperación entre ellos.

En el fútbol, teniendo como referencia la posesión de la pelota se pueden diferenciar dos fases y en cada una tres principios

Ataque: cuando el equipo tiene la posesión de la pelota

- Meter gol
- Avanzar
- Mantener la posesión de la pelota

Defensa: cuando el equipo no tiene la posesión de la pelota

- Proteger la porteria
- Cortar el ataque del rival
- Recuperar la pelota

Ante las situaciones que se plantean en el fútbol, cada jugador debe responder de una u otra manera. La efectividad del juego reside en elegir la más idónea de cuantas posibilidades de respuesta se presentan. Esas respuestas o comportamientos serán diferentes si el jugador está atacando o defendiendo.

Estas son las posibilidades que se les presentan a los jugadores de campo tanto en ataque como en defensa:

<table>
<tr><td colspan="2">Jugador de campo</td></tr>
<tr><td>Ataque</td><td>Defensa</td></tr>
<tr><td>Con pelota</td><td rowspan="4">Obstaculizar el remate
Interceptar
Achicar el espacio
Entrada
Repliegue
Marcaje
Cobertura
Permuta
Basculación
Retardar el ataque
Ubicarse según el sistema</td></tr>
<tr><td>Rematar
Avanzar
Pasar
Superar al rival
Fintar
Ubicarse según el sistema
Temporizar</td></tr>
<tr><td>Sin Pelota</td></tr>
<tr><td>Avanzar
Dar opción de pase
Ubicarse según el sistema
Ampliar el espacio
Buscar el espacio libre
Desmarcarse
Atraer al defensor
Ayudar a quien tiene la pelota
Aumentar el espacio</td></tr>
</table>

Figura 67: Tabla de comportamientos de un jugador de campo

Estas son las posibilidades que se les presentan a los porteros:

Portero	
Ataque	**Defensa**
Pasar Fintar Temporizar Dar opción de pase	Interceptar Colocación Salida

Figura 68: Tabla de comportamientos de un portero

5.1.1 Comportamientos

Rematar

Consiste en realizar cualquier lanzamiento con intención de meter gol.

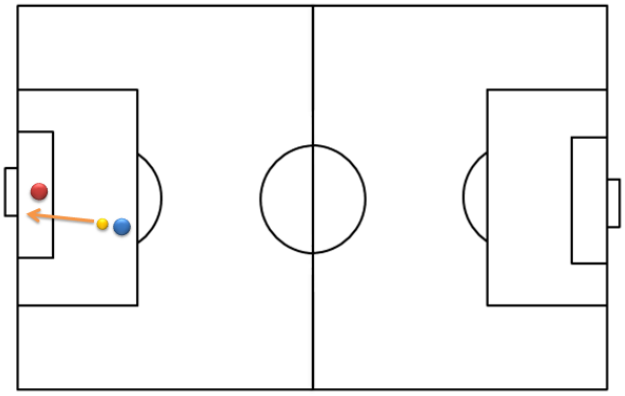


Figura 69: Ilustración del comportamiento "Rematar"

Avanzar con pelota

El objetivo es llevar la pelota lo más cerca de la portería contraria.

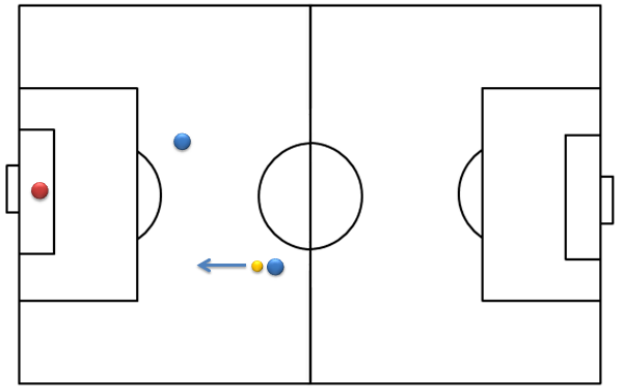


Figura 70: Ilustración del comportamiento “Avanzar con pelota”

Pasar

Relación que se da entre dos futbolistas del mismo equipo por medio de la pelota en la que un futbolista chuta la pelota a otro futbolista.

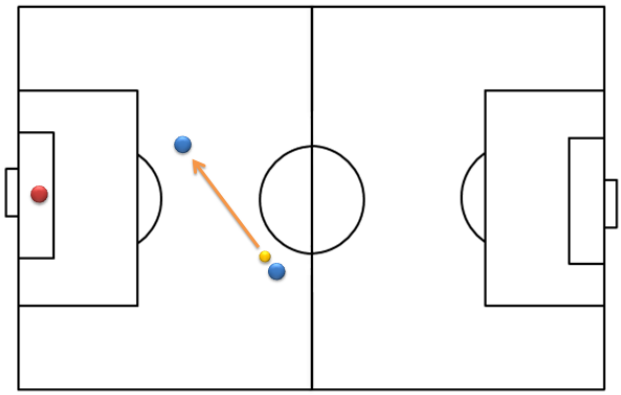


Figura 71: Ilustración del comportamiento "Pasar"

Superar al rival

Cuando teniendo la pelota controlada y dirigiéndose hacia la portería contraria se deja al rival detrás.

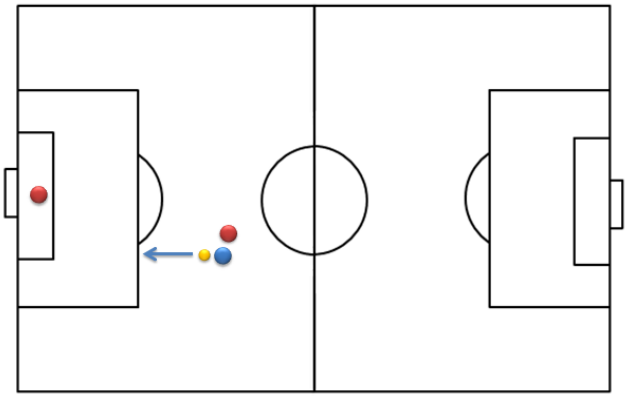


Figura 72: Ilustración del comportamiento "Superar al rival"

Finta

Engañar al contrario por medio de un amago. Dar a entender que se va a realizar una acción y luego realizar otra.

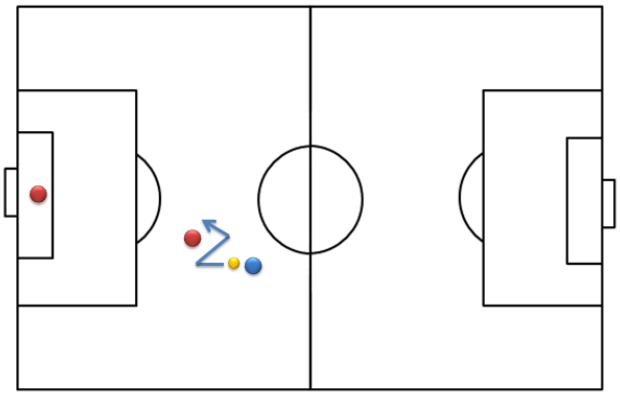


Figura 73: Ilustración del comportamiento "Finta"

Temporizar

Ralentizar el movimiento de la pelota para tener ventaja tácticamente y para que pase el tiempo.

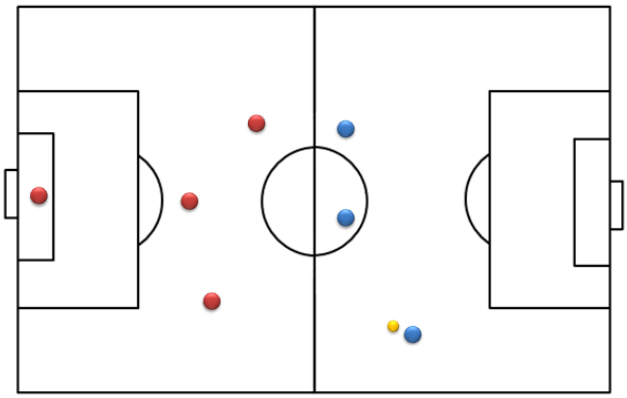


Figura 74: Ilustración del comportamiento "Temporizar"

Avanzar sin pelota

Acompañar al futbolista que lleva la pelota hacia la portería contraria en sentido directo para darle opción de pase.

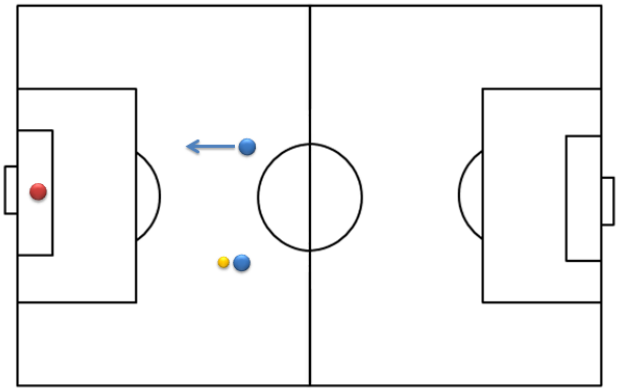


Figura 75: Ilustración del comportamiento "Avanzar sin pelota"

Dar opción de pase

Buscar la mejor situación en el campo para que el futbolista con la posesión de la pelota tenga opción de pasarla.

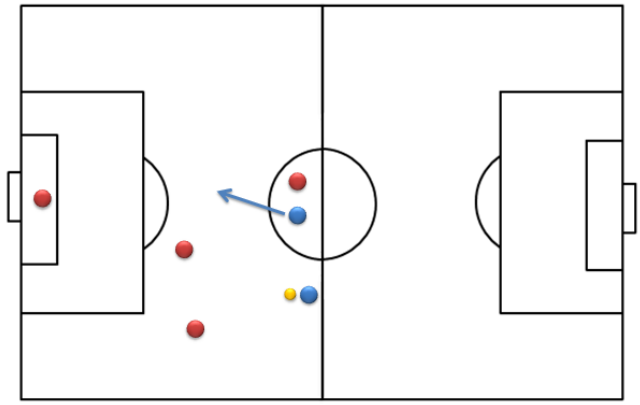


Figura 76: Ilustración del comportamiento "Dar opción de pase"

Ayudar a quien tiene la pelota

Acción que se realiza para ayudar a un futbolista en apuros.

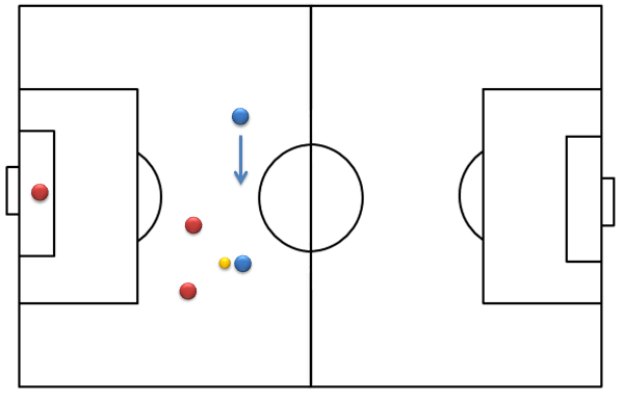


Figura 77: Ilustración del comportamiento "Ayudar a quien tiene la pelota"

Ubicarse según el sistema

Situarse en la posición fija que se tiene según el sistema de juego del equipo.

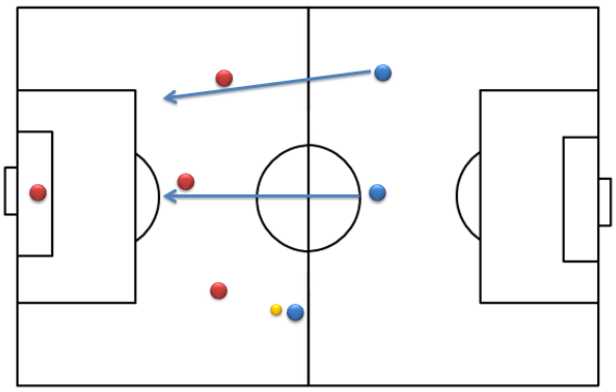


Figura 78: Ilustración del comportamiento "Ubicarse según sistema"

Ampliar el espacio

Ensanchar y alargar la zona de juego por medio de movimientos individuales o colectivos.

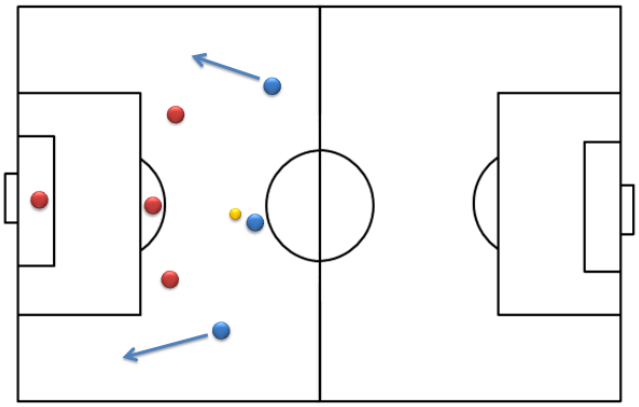


Figura 79: Ilustración del comportamiento "Ampliar el espacio"

Buscar el espacio libre

Si un futbolista con un movimiento concreto se va de un espacio, ese espacio quedara libre para que pueda ser ocupado por otro futbolista.

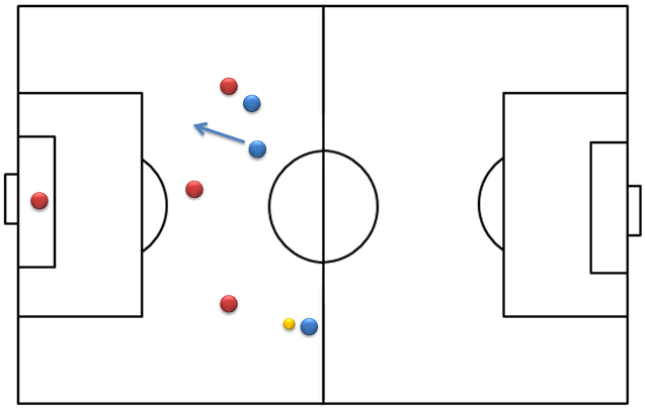


Figura 80: Ilustración del comportamiento "Buscar el espacio libre"

Desmarcarse

Escaparse de la vigilancia directa del rival.

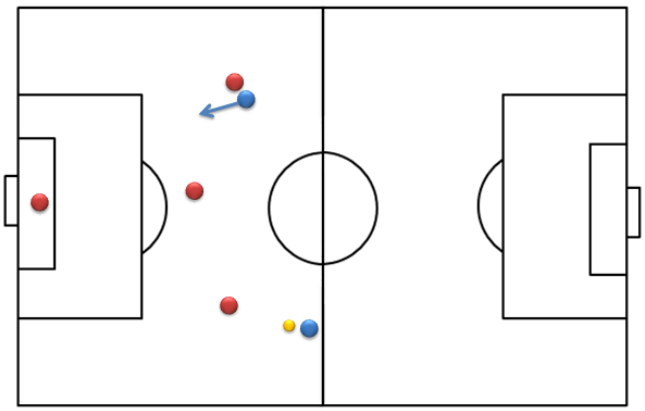


Figura 81: Ilustración del comportamiento "Desmarcarse"

Atraer al defensor

Atraer mediante diversos movimientos al defensor de manera que el espacio libre que se crea puede ser aprovechado por otro de los futbolistas.

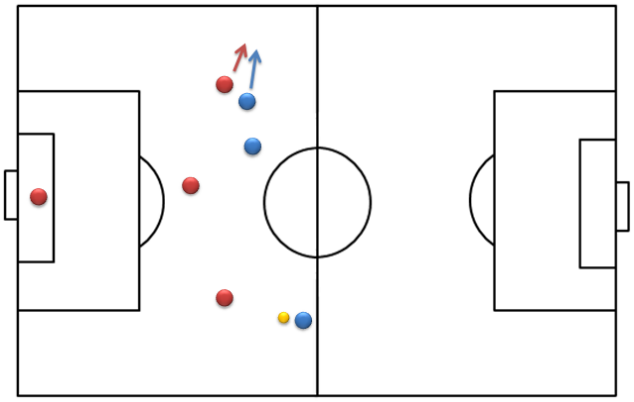


Figura 82: Ilustración del comportamiento "Atraer al defensor"

Obstaculizar el remate

No dejar al contrario que realice ningún tipo de tiro con intención de meter gol o hacer que lo realice con dificultad.

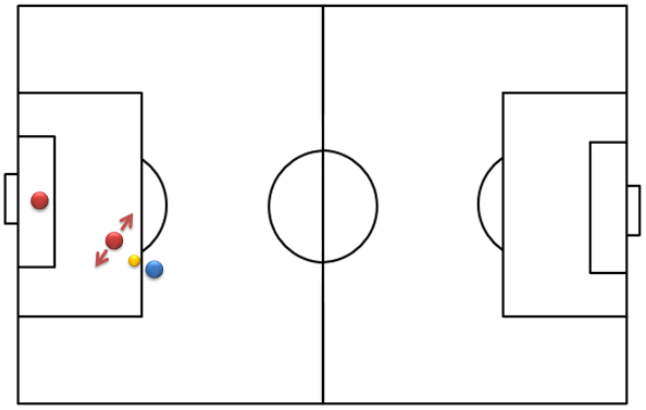


Figura 83: Ilustración del comportamiento "Obstaculizar el remate"

Entrada

Conjunto de acciones para robar la pelota al contrario.

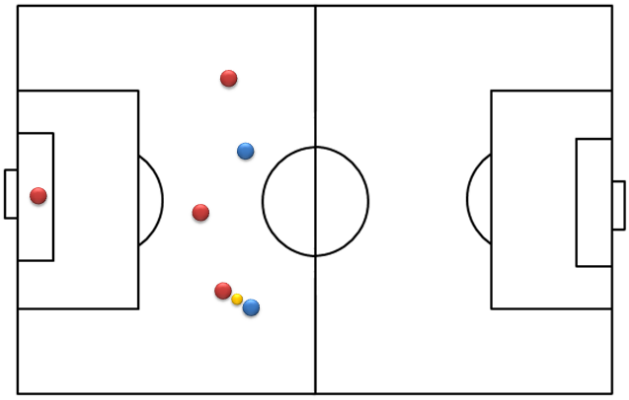


Figura 84: Ilustración del comportamiento "Entrada"

Impedir el avance

No dejar que los rivales se acerquen a la portería.

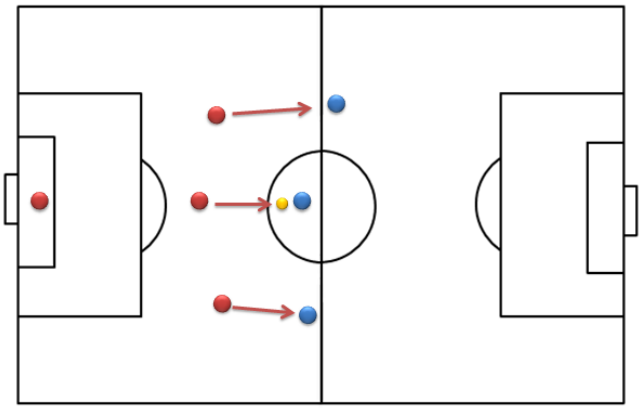


Figura 85: Ilustración del comportamiento "Impedir el avance"

Achique de espacios

Disminuir el espacio de juego efectivo o peligroso del equipo rival.

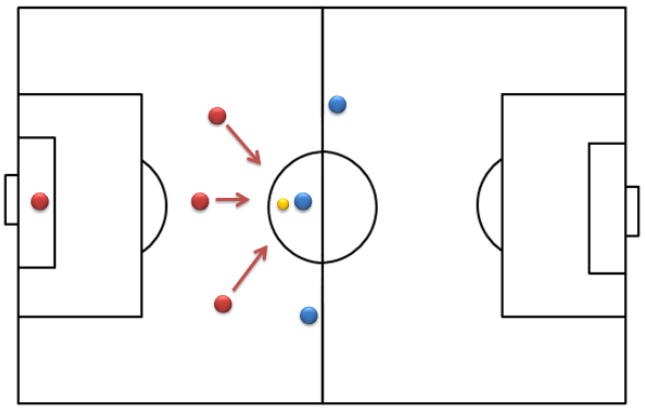


Figura 86: Ilustración del comportamiento "Achique de espacios"

Repliegue

Movimientos que se realizan para volver a las posiciones defensivas cuanto antes, una vez que se ha perdido el balón en ataque.

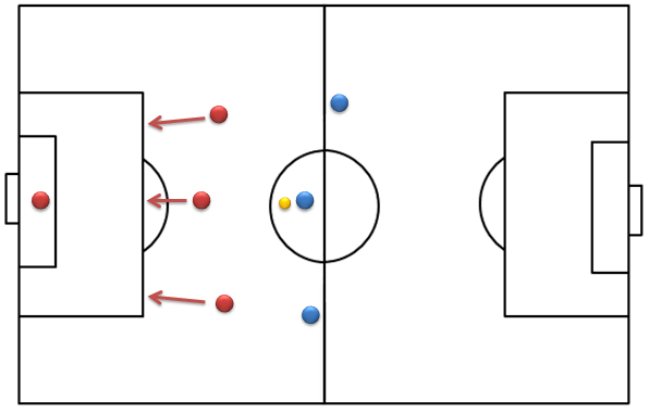


Figura 87: Ilustración del comportamiento "Repliegue"

Marcaje

Todas las acciones que hacen los futbolistas del equipo sobre un rival.

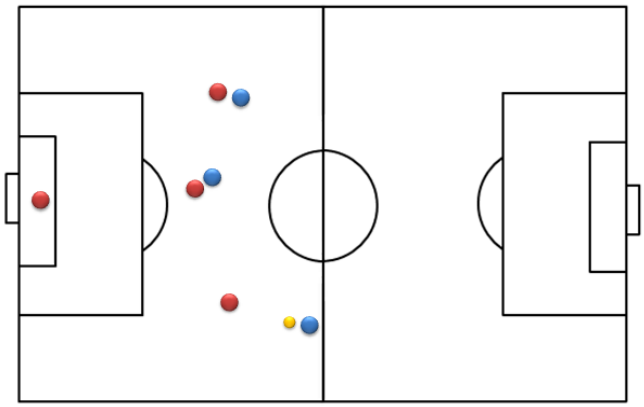


Figura 88: Ilustración del comportamiento "Marcaje"

Cobertura

Estar bien ubicado para ayudar a un futbolista del equipo, cubriéndole las espaldas, si un rival teniendo la pelota tiene posibilidades de superarle.

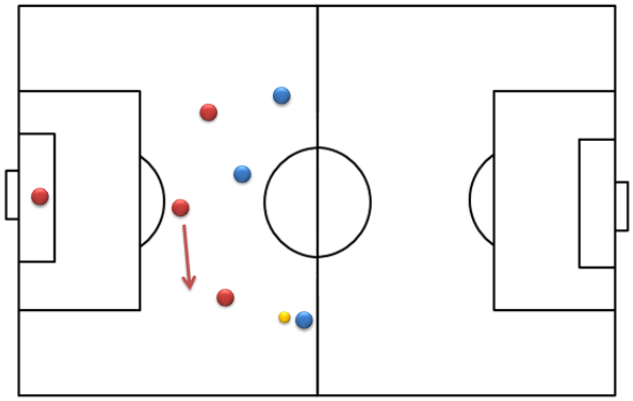


Figura 89: Ilustración del comportamiento "Cobertura"

Permuta

Si el rival con la pelota ha superado a un futbolista, otro futbolista sale de su posición para ayudarle, cubriendo el espacio que ha dejado libre.

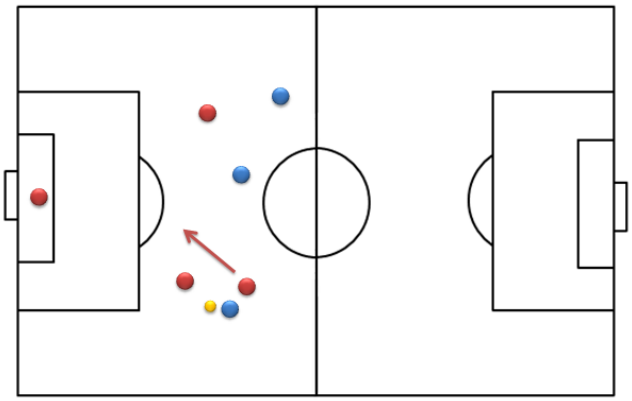


Figura 90: Ilustración del comportamiento "Permuta"

Basculación

Amoldar la ubicación dependiendo de donde estén la pelota y el resto de futbolistas del equipo.

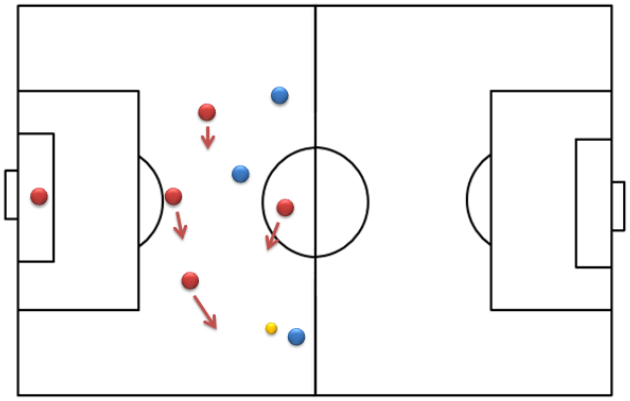


Figura 91: Ilustración del comportamiento "Basculación"

Interceptar

Cualquier tipo de acción que realizamos para que la pelota lanzada por el contrario no llegue a su sitio o a un receptor.

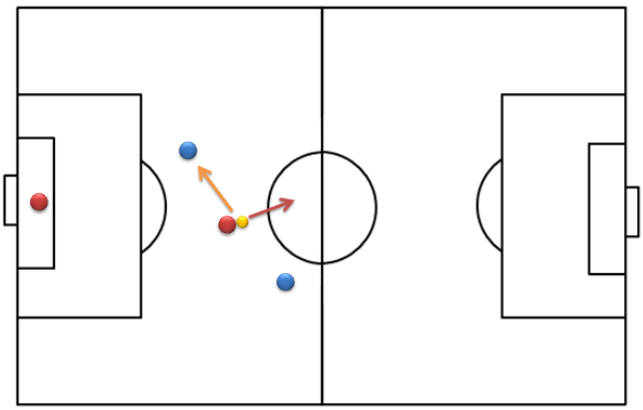


Figura 92: Ilustración del comportamiento "Interceptar"

Retardar el ataque

Acción que se realiza cuando, teniendo la pelota el contrario, en lugar de entrarle se da tiempo para que llegue la ayuda de otros futbolistas del equipo.

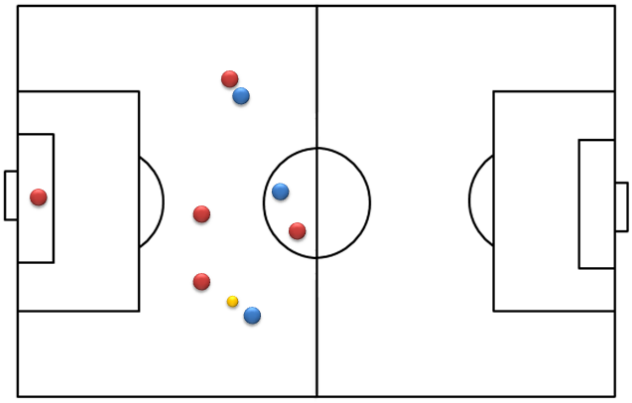


Figura 93: Ilustración del comportamiento "Retardar el ataque"

Colocación

Conjunto de movimientos que realiza el portero para cubrir lo mejor posible la portería.

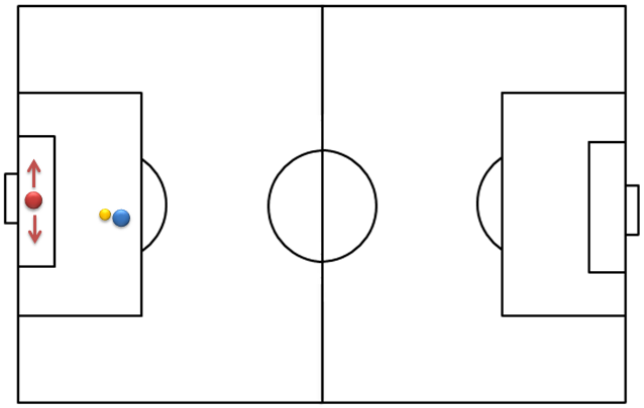


Figura 94: Ilustración del comportamiento "Colocación"

Salida

Movimiento que realiza un portero hacia el jugador que tiene la pelota para reducir su ángulo de remate.

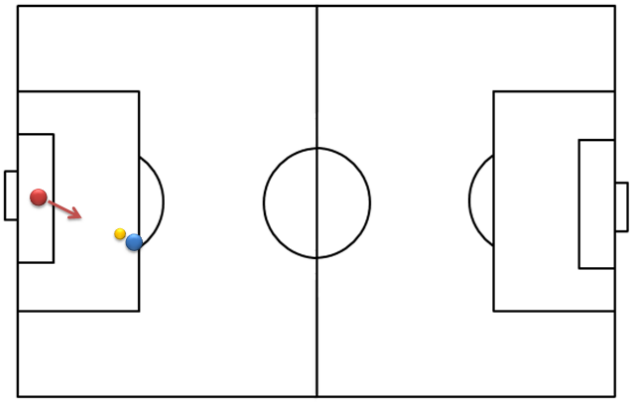


Figura 95: Ilustración del comportamiento "Salida"

5.1.2 Roles

Portero

Es la posición que representa la última línea de defensa entre el ataque del oponente y la propia portería. La función principal del jugador en esta posición es la de defender directamente la portería y evitar que el contrario marque gol.

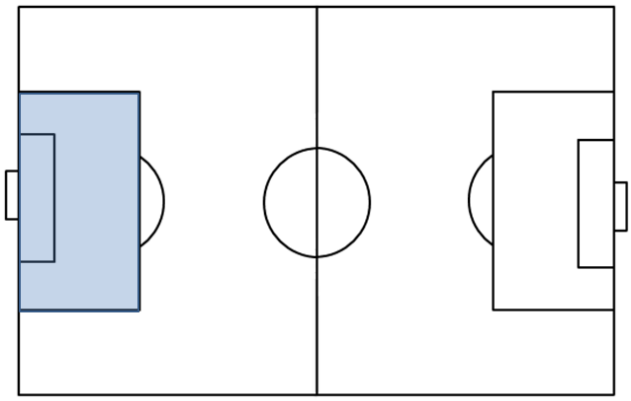


Figura 96: Regiones de influencia de un portero

Laterales

Son los defensas que se encargan de ir por la banda hasta la mitad del campo de fútbol o hasta la otra portería pero deben volver y defender mientras el otro equipo ataca.

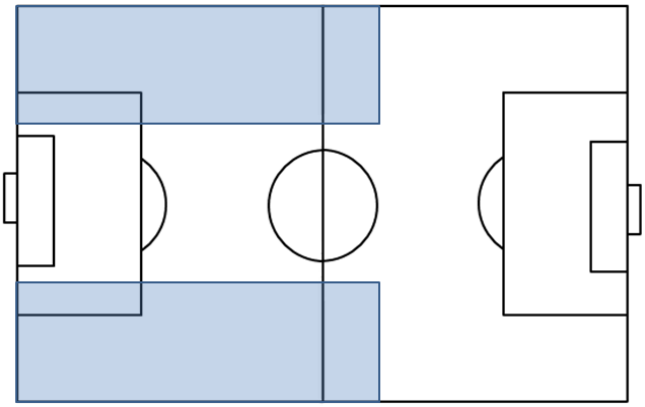


Figura 97: Regiones de influencia de los laterales

Centrales

Son los defensas que se ubican justo enfrente de la portería y tienen una función puramente defensiva, se les denomina "último hombre" al ser los últimos defensas que protegen la portería.

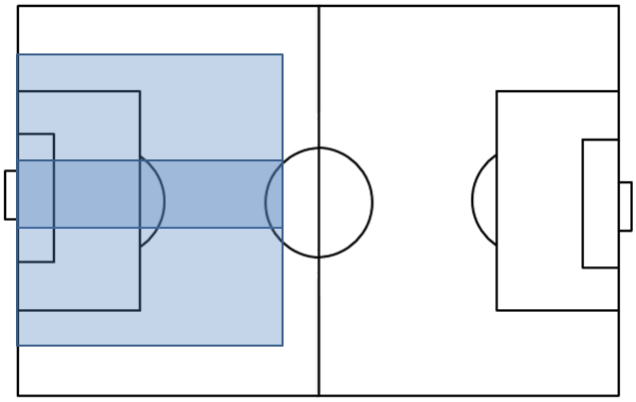


Figura 98: Regiones de influencia de los centrales

Medio centro

La función principal del medio centro es recuperar balones y cortar el avance ofensivo del equipo contrario manteniéndolos lejos de la defensa y ayudándoles constantemente.

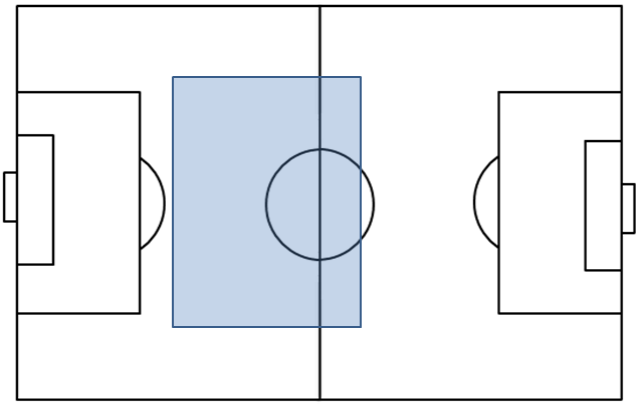


Figura 99: Regiones de influencia de un medio centro

Interiores

La función principal de los interiores es utilizar las bandas hacia la portería contraria y desbordar con centros así como ayudar defensivamente en el caso de que el equipo contrario ataque.

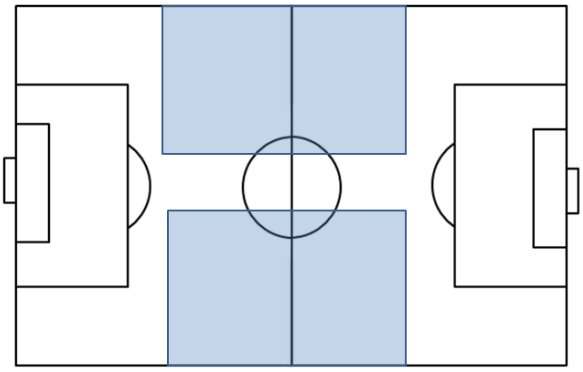


Figura 100: Regiones de influencia de los interiores

Media punta

Es el centrocampista más adelantado y se sitúa por detrás de los delanteros. Su función es la de coordinar el ataque del equipo, dar el último pase a los delanteros o aprovecharse de los huecos libres que deja el defensa rival.

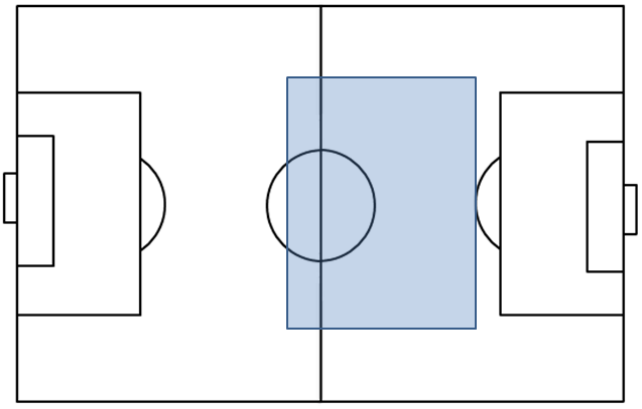


Figura 101: Regiones de influencia de los media punta

Extremos

Los extremos son jugadores con gran movilidad que retroceden varios metros del área rival para apoyar a los centrocampistas y recuperar el balón así como desbordar por las bandas y acompañar al delantero centro.

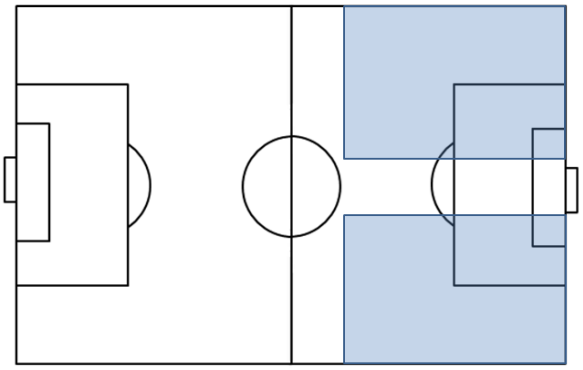


Figura 102: Regiones de influencia de los extremos

Delantero centro

Los delanteros centro son usualmente los encargados de marcar goles de un equipo. Su función es la de meter la pelota dentro de la portería, por lo que la puntería, la potencia y el remate de cabeza son las características más buscadas en estos jugadores.

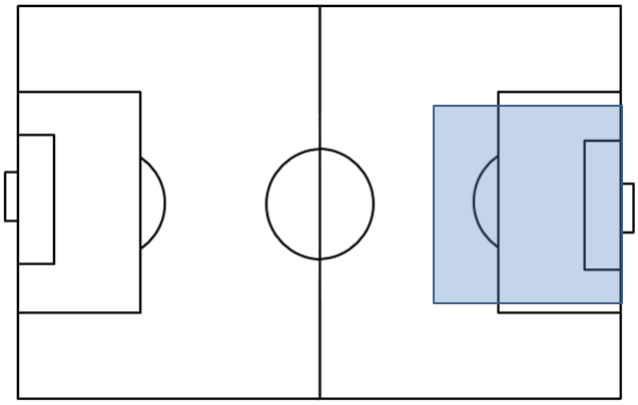


Figura 103: Regiones de influencia del delantero centro

5.1.3 Los futbolistas como agentes inteligentes

El entorno

El entorno de un futbolista está compuesto por el campo de fútbol, las porterías, la pelota y el resto de futbolistas.

Este entorno es accesible ya que los futbolistas son capaces de percibir su estado completo. En todo momento conocen las acciones que están realizando el resto de futbolistas así como la posición y dirección de la pelota.

No es determinista porque no es posible conocer a partir del estado actual del entorno y la decisión tomada por el futbolista, el estado futuro del propio entorno ni del futbolista. Por ejemplo, un futbolista puede tomar la decisión de atrapar la pelota pero puede fallar en su intento porque su rival se lo impide.

No es episódico ya las decisiones tomadas por los futbolistas pueden afectar a decisiones futuras. Si el futbolista que posee la pelota toma la decisión de pasarla a otro futbolista, la decisión que realizará después podrá ser diferente a si no la hubiera pasado al compañero.

Se trata de un entorno dinámico ya que su estado no es solo alterado por la acción de un futbolista sino por la de varios, de manera que es complicado hacer una planificación de las acciones a realizar ya que el estado se altera continuamente.

Por último, se trata de un entorno continuo porque no es posible concretar todos sus estados. Por ejemplo, todos los jugadores pueden moverse a cualquier posición del campo por lo que será necesario realizar una discretización del campo para simplificar la toma de decisiones.

Las percepciones

Al tratarse de un entorno accesible, los futbolistas perciben en todo momento la posición, orientación y velocidad del resto de futbolistas y de la pelota. A través de dichas percepciones el futbolista puede por ejemplo deducir si se encuentra en peligro porque un rival se está acercando o si se encuentra a una distancia suficiente para poder chutar a portería.

Las acciones

Las acciones que puede realizar un futbolista en el videojuego para modificar su entorno son ir a una posición, orientarse para mirar a una posición, atrapar la pelota para hacer suya la posesión y chutar la pelota

La toma de decisiones

La toma de decisiones para un futbolista consiste en dado el conocimiento que tiene del entorno decidir en todo momento hacia donde debe ir y hacia donde debe mirar. En el caso de tener la posesión de la pelota decidir hacia donde debe chutar y en el caso de no tenerla decidir si puede o no atrapar la pelota.

5.2 Diseño

Para poder demostrar el funcionamiento del motor de inteligencia artificial aplicado en un videojuego de fútbol se ha decidido desarrollar sólo un subconjunto de los comportamientos que pueden realizar los futbolistas así como una táctica colectiva para cada uno de los tipos de coordinación que soporta el motor. Al no pretenderse implementar un videojuego de fútbol completo se han omitido también por ejemplo situaciones como córners, penaltis o faltas.

5.2.1 Representación del conocimiento

Para facilitar la toma de decisiones de los futbolistas se ha dividido el campo en regiones permitiendo asociar una determinada posición a una región. En concreto se han creado 132 regiones tal y como se muestran en la siguiente ilustración.

Figura 104: Captura de pantalla del videojuego donde se muestra la división del campo en regiones

La división del campo en regiones permite definir fácilmente las zonas donde tiene influencia cada futbolista en función de su rol así como identificar en que región se encuentra la pelota.

En función de la región en la que se encuentra la pelota se pueden diferenciar tres fases en el desarrollo del ataque de un equipo. Tomando como referencia al equipo que ataca hacia la portería derecha, se define:

- **Zona de iniciación** (rojo). Es la zona donde se inicia el juego y los futbolistas se van situando.
- **Zona de creación** (amarillo). Es la zona donde se crea el juego ofensivo y se busca una opción de gol.
- **Zona de finalización** (azul). Es la zona donde se busca la finalización de las jugadas con un remate a portería.

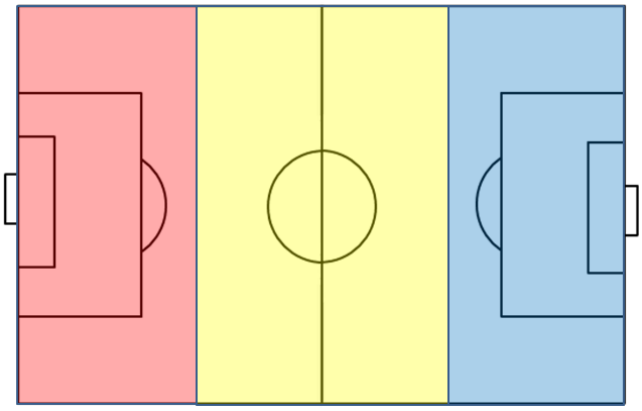


Figura 105: Ilustración de las fases en el desarrollo del ataque de un equipo

La siguiente tabla muestra todo el conocimiento tanto individual como colectivo de los futbolistas y que les permite realizar la toma de decisiones más adecuada:

Individual	**Colectivo**
Los jugadores del equipo propio	El equipo que posee la pelota
Los jugadores del equipo rival	El último equipo que ha tenido la posesión
La portería propia	El jugador que posee la pelota
La portería rival	El último jugador que ha tenido la posesión
Las regiones donde tiene influencia	La región donde está la pelota
Región actual	El jugador que ha chutado la pelota
Distancia a la pelota	El jugador que debe recibir un pase
Distancia a portería rival	El último jugador que ha recibido un pase
Si el equipo tiene la posesión de la pelota	
Si el jugador tiene la posesión de la pelota	
Si es el jugador más cercano a la pelota	
Si se está en rango de atrapar la pelota	
Si se está en rango de parar la pelota	
Si se está en rango de rematar a portería	
Si ha chutado la pelota	
Si se ha chutado la pelota	
Si es receptor de un pase	
Si está en peligro	
Si esta en inferioridad	
Si la pelota está en zona de iniciación	

Si la pelota está en zona de creación Si la pelota está en zona de finalización Target movimiento Target orientación Target de chute Posición de movimiento Posición de orientación	

Figura 106: Tabla con el conocimiento individual y colectivo

5.2.2 Árbol de objetivos

Jugador de campo

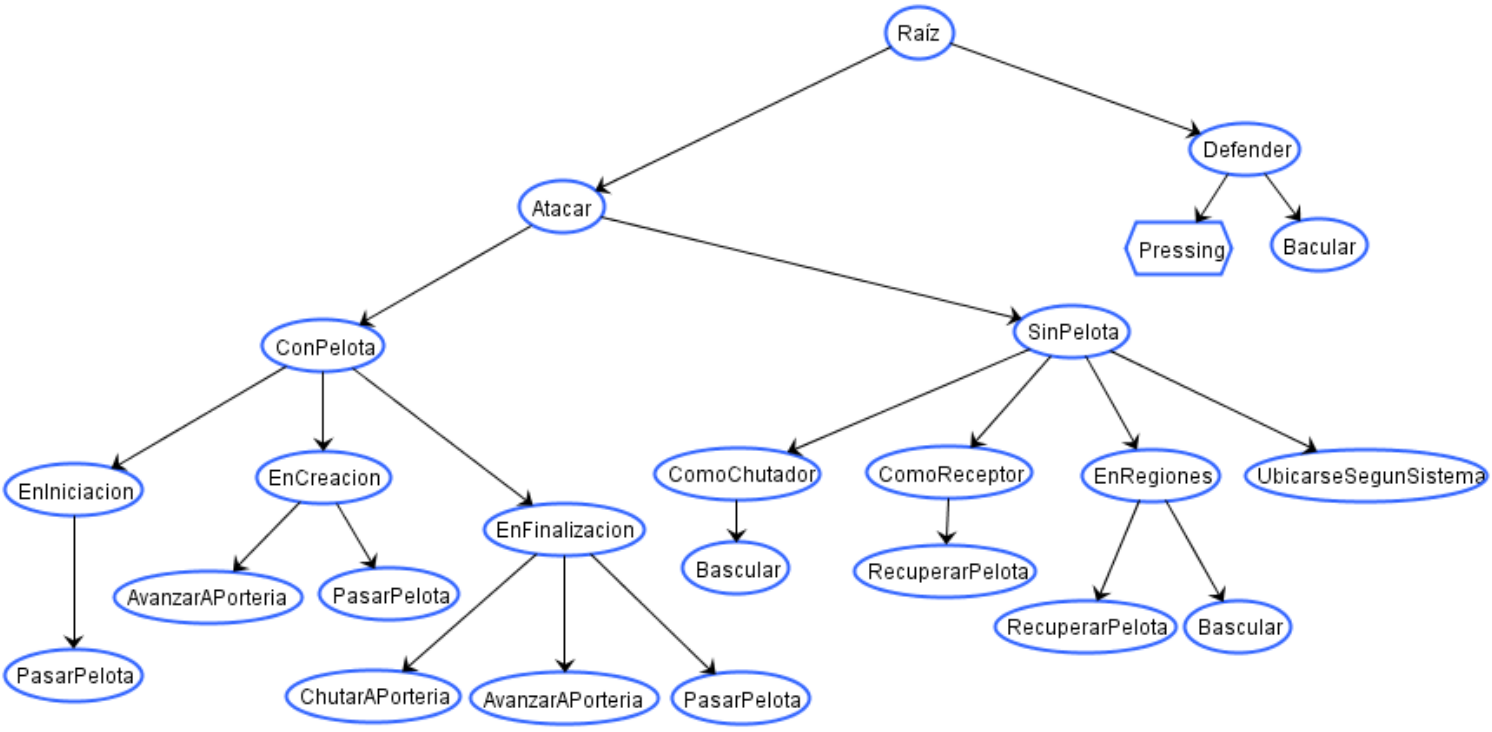


Figura 107: Árbol de objetivos de un jugador de campo

Si el equipo del futbolista tiene la posesión de la pelota entonces su objetivo es atacar y si no es defender.

La manera en como ataca un futbolista depende de si tiene la pelota o no. Si tiene la pelota se comportará de una forma distinta en función de la zona en la que se encuentre la pelota.

Si la pelota se encuentra en la fase de iniciación el futbolista intentará pasar la pelota lo antes posible para alcanzar posiciones ofensivas lo antes posible. Si se encuentra en fase de creación, el futbolista intentará avanzar con la pelota el máximo de metros posibles hasta que se encuentre en una situación de peligro donde sería más conveniente pasarle la pelota a otro compañero. Si por el contrario la pelota se encuentra en fase de finalización, el futbolista intentará rematar a portería tan pronto como le sea posible, si no le es posible rematar a portería porque se encuentra muy lejos intentará acercarse pero si se encuentra en inferioridad numérica la mejor decisión será encontrar un compañero mejor situado.

Un futbolista aunque no tenga la pelota durante un ataque también tiene comportamientos ofensivos. Si la pelota está en sus regiones de influencia definidas por el rol que desempeña en el equipo y ningún futbolista posee en ese momento la pelota y es el más cercano a la pelota entonces irá a recuperarla. Este caso se produce cuando por ejemplo se ha rematado a portería y la pelota ha rebotado en el poste el equipo del futbolista que ha rematado tiene la posesión de la pelota pero en ese momento ningún futbolista la posee. Si no se produce dicha situación el futbolista intentara bascular respecto a la pelota y al resto de compañeros para avanzar hacia la portería y situarse a una distancia suficiente para apoyar al compañero que conduce la pelota.

En el caso que la pelota se encuentre fuera de la región de influencia quiere decir que el futbolista se encuentra mal situado respecto de la formación del equipo y por lo tanto debe ubicarse en su región de influencia más cercana.

Los casos en que se ataca como chutador o receptor son unos casos especiales que permiten coordinar a los futbolistas cuando se realiza un pase y serán vistos en el apartado de tácticas colectivas.

Por último, cuando el equipo no tiene posesión es necesario defenderse, las opciones que tiene un futbolista son o participar en la táctica colectiva de pressing que consiste en presionar colectivamente al rival para recuperar la pelota o bien bascular para retroceder defensivamente a medida que se acerca la pelota a la portería y mantener una buena formación defensiva.

Portero

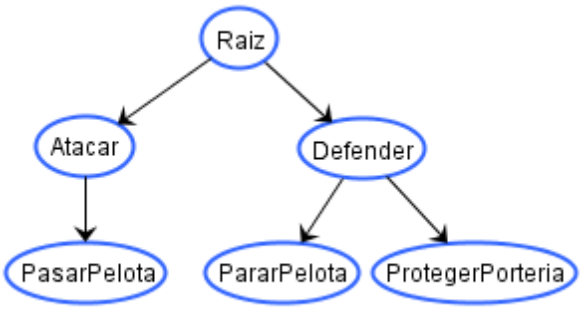


Figura 108: Árbol de objetivos de un portero

El portero se comporta diferente en función de si tiene la posesión de la pelota o no la tiene.

Si tiene la posesión de la pelota debido a un pase o porque la ha parado tras un remate entonces lo que hará será pasarla rápidamente a un compañero para iniciar el ataque.

En el momento que no tiene la posesión estará alerta ante cualquier posible remate para que en el caso que de producirse pueda pararlo. Mientras no se realiza ningún remate se colocará debidamente ante la portería y la pelota para así poder reducir el ángulo de remate.

5.2.3 Comportamientos individuales

Avanzar a portería

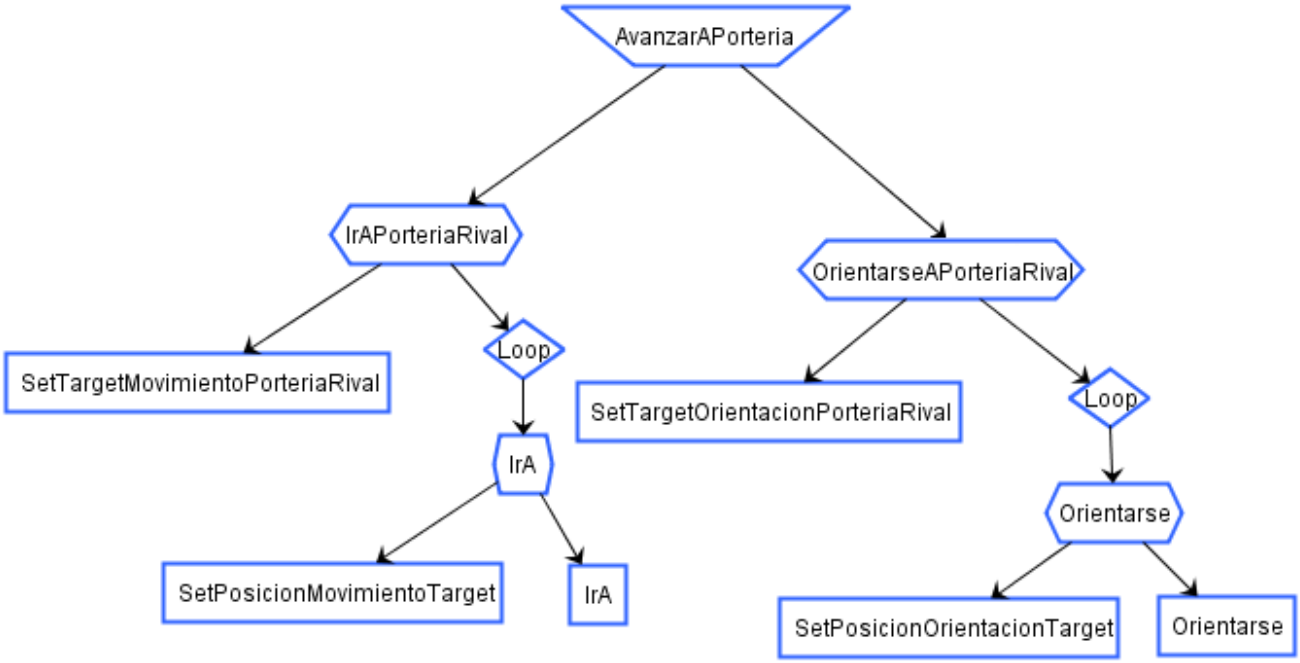


Figura 109: Árbol del comportamiento "Avanzar a portería"

El comportamiento de avanzar a portería consiste en ir hacia la portería rival y orientarse hacia la dirección de movimiento.

Para ir a la portería rival es necesario modificar el conocimiento del futbolista indicando que el target de movimiento va a ser la portería rival y posteriormente acercarse en cada iteración la posición del target mediante el sub-comportamiento "irA" que extrae la posición del target y ejecuta la acción "irA" que mueve al futbolista.

Paralelamente al movimiento del futbolista se realiza el cambio de su orientación o lo que es lo mismo, hacia donde mira. Lo primero que hará será modificar su conocimiento con la portería como target de orientación, seguidamente en cada iteración el futbolista modificará su orientación mediante el comportamiento "Orientarse" que extrae la posición del target y ejecuta la acción orientarse del futbolista.

Recuperar la pelota

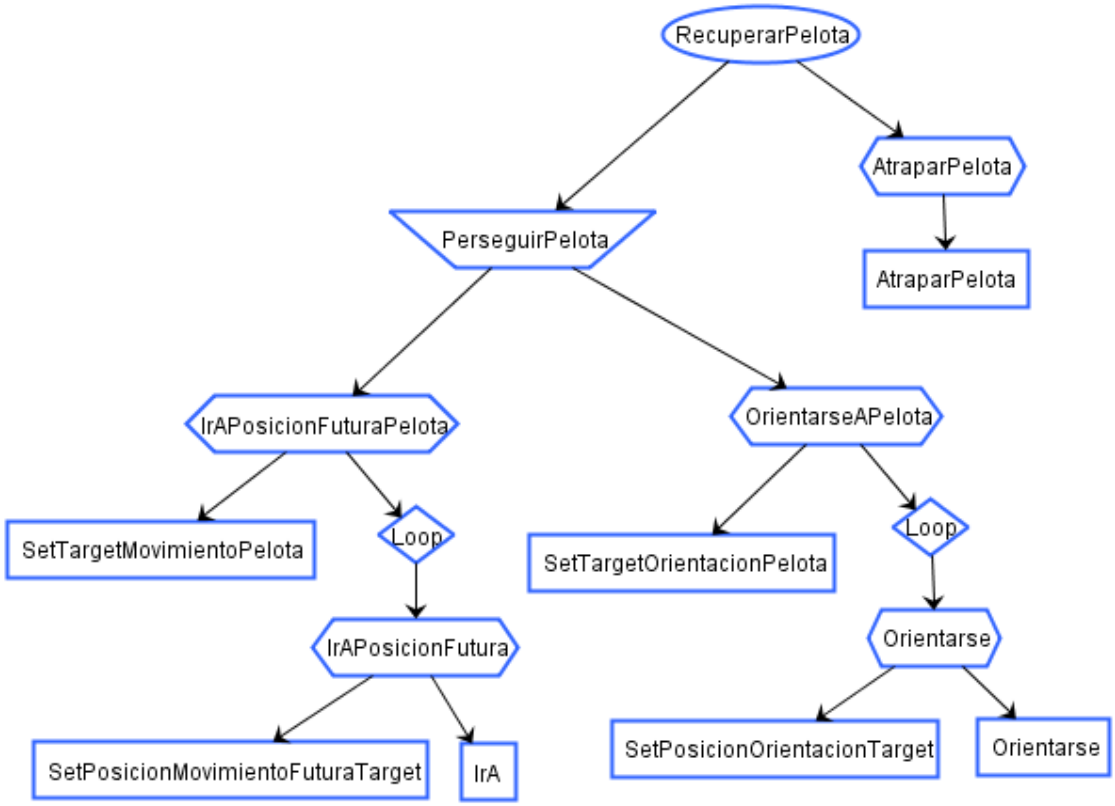


Figura 110: Árbol del comportamiento "Recuperar la pelota"

El comportamiento de recuperar la pelota consiste en perseguir la pelota hasta encontrarse los suficientemente cerca para poder atraparla.

Para poder perseguir la pelota es necesario anticiparse al movimiento de la misma haciendo una predicción de donde estará unos instantes en el futuro. Para ello el sub-comportamiento “irAPosicionFuturaPelota” lo primero que hace es modificar el conocimiento del futbolista con la pelota como target y posteriormente ejecutar indefinidamente el comportamiento “IrAPosicionFutura” que calcula la posición futura del target y se mueve hacia él.

Paralelamente, el futbolista mantendrá la vista siempre fijada en la pelota por lo que lo primero que hará será modificar su conocimiento para marcarla como target de orientación. Como se trata de un target dinámico ya que la pelota estará continuamente en movimiento, el sub-comportamiento “Orientarse” se encarga de obtener su posición y de orientar al futbolista los grados necesarios.

Chutar a portería

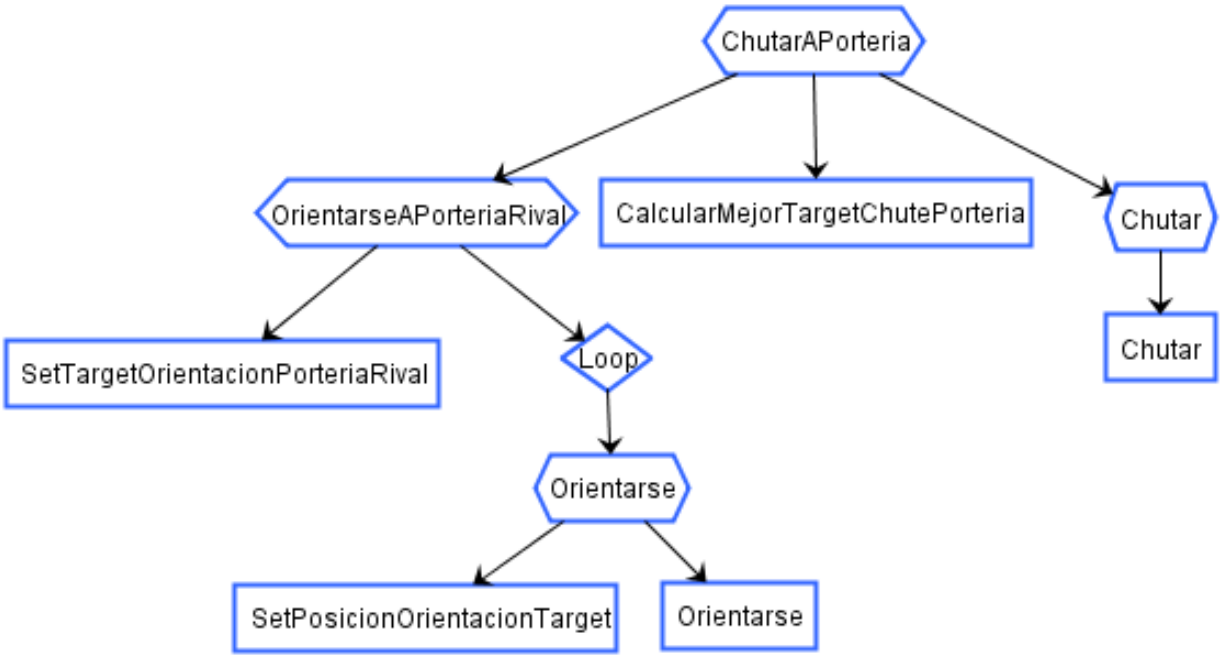


Figura 111: Árbol del comportamiento "Chutar a portería"

Si el futbolista decide que debe chutar a portería, lo primero que debe hacer es orientarse a la portería.

El sub-comportamiento “OrientarseAPorteriaRival” es el mismo que se utiliza en el comportamiento “AvanzarAPorteria” por lo que sirve de ejemplo de la reutilización de los sub-comportamientos para crear una gran variedad de comportamientos.

Posteriormente calculará cuál es la mejor posición dentro de la portería a la que debe chutar para poder marcar gol, y por ultimo realizar el chute a dicha posición.

Pasar la pelota

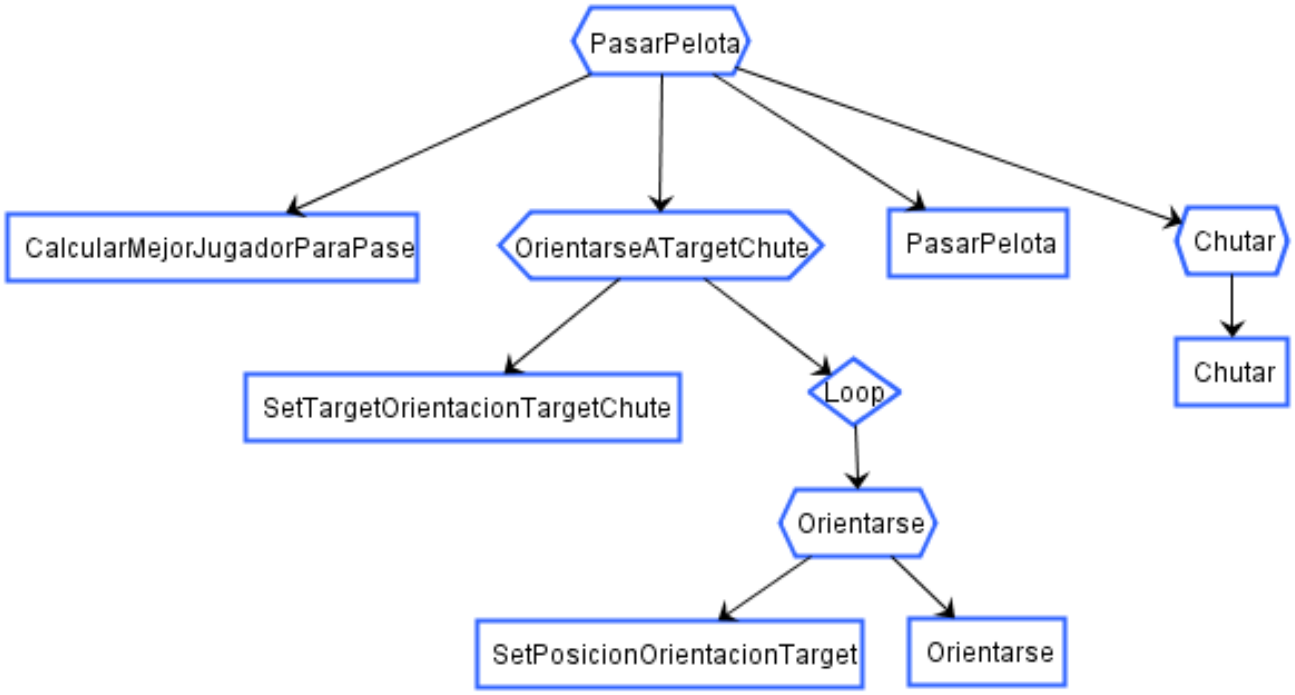


Figura 112: Árbol del comportamiento "Pasar la pelota"

Para que un futbolista realice un pase lo primero que debe hacer es decidir a quién se lo va a realizar. La acción "CalcularMejorJugadorParaPase" evalúa a los compañeros del futbolista y selecciona a aquel que esté mejor situado.

Una vez ha seleccionado el futbolista al que se le va a hacer el pase se tiene que orientar hacia él. Cuando ya se encuentra correctamente orientado modifica el comportamiento compartido indicando a quien se le va a hacer el pase mediante la acción "PasarPelota". Esta notificación se indica después de haberse orientado para evitar la situación en que el futbolista que va a recibir la pelota inicie el comportamiento de recibir la pelota cuando aun no se ha realizado el pase. Por último, el futbolista ejecuta la acción de chutar para hacer efectivo el pase.

Ubicarse según el sistema

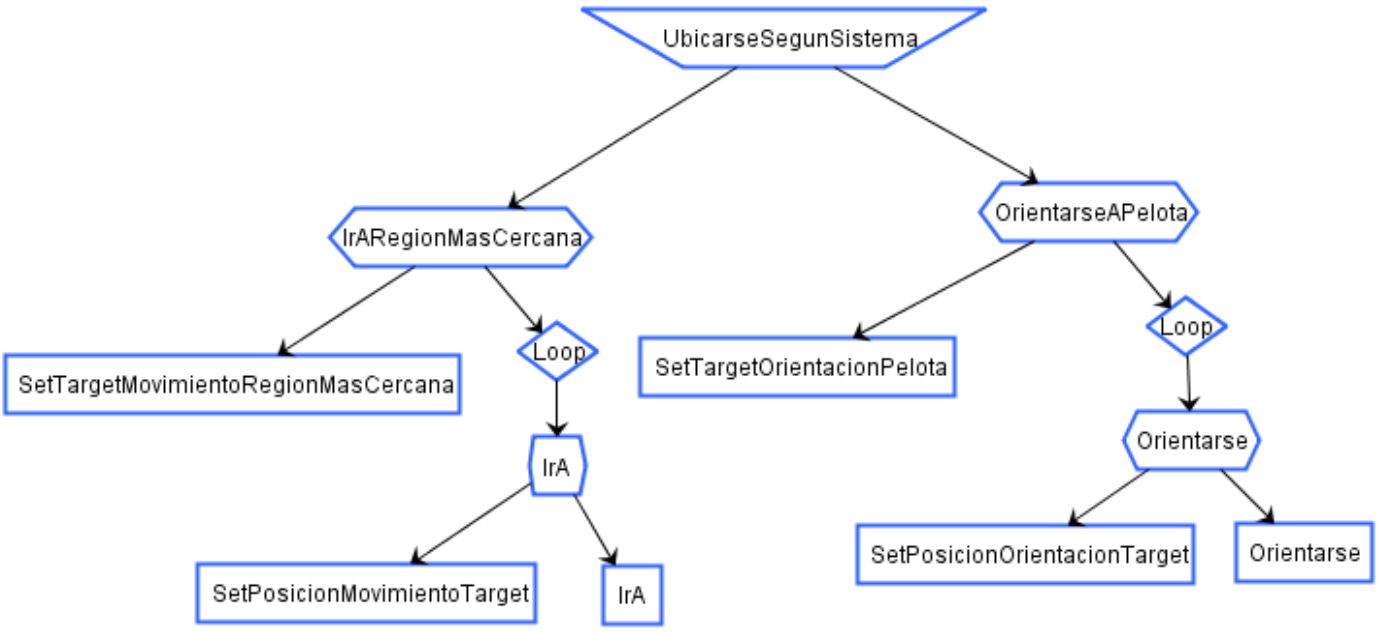


Figura 113: Árbol del comportamiento "Ubicarse según el sistema"

La finalidad de este comportamiento es situar al futbolista dentro de las regiones que tiene asignadas en función de su rol.

Su nodo raíz es un comportamiento paralelo que permite ejecutar a la vez los sub-comportamientos “IrARegionMasCercana” y “OrientarseAPelota”.

El sub-comportamiento “IrARegionMasCercana” tiene como objetivo situar al futbolista en la región más cercana a la posición en la que se encontraba en el momento de iniciar el comportamiento. Para ello primero calcula la región más cercana, obtiene su posición central y luego acerca al futbolista hacia esa posición.

El sub-comportamiento “OrientarseAPelota” tiene como objetivo modificar la orientación del futbolista de manera que mire en todo momento a la pelota que está en movimiento. Para ello primero fija a la pelota como entidad a seguir y posteriormente obtiene su posición para poder orientarse hacia ella.

Bascular

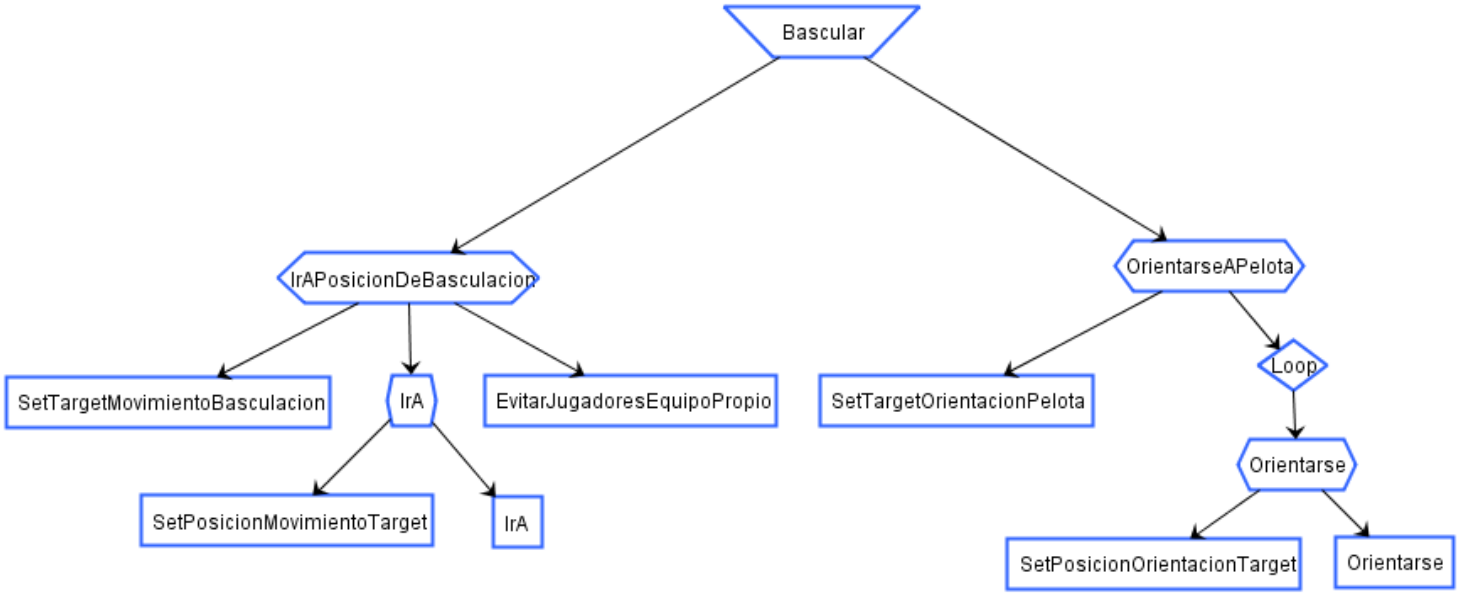


Figura 114: Árbol del comportamiento "Bascular"

La finalidad de este comportamiento es amoldar la ubicación del futbolista dependiendo de donde estén la pelota y el resto de futbolistas del equipo.

Su nodo raíz es un comportamiento paralelo que permite ejecutar a la vez los sub-comportamientos “IrAPosicionDeBasculacion” y “OrientarseAPelota”.

El sub-comportamiento “IrAPosicionDeBasculacion” tiene como objetivo situar al futbolista en región propia más cercana a la pelota a la vez que mantiene una cierta distancia con los compañeros más cercanos.

El sub-comportamiento “OrientarseAPelota” tiene como objetivo modificar la orientación del futbolista de manera que mire en todo momento a la pelota que está en movimiento. Para ello primero fija a la pelota como entidad a seguir y posteriormente obtiene su posición para poder orientarse hacia ella.

Parar la pelota

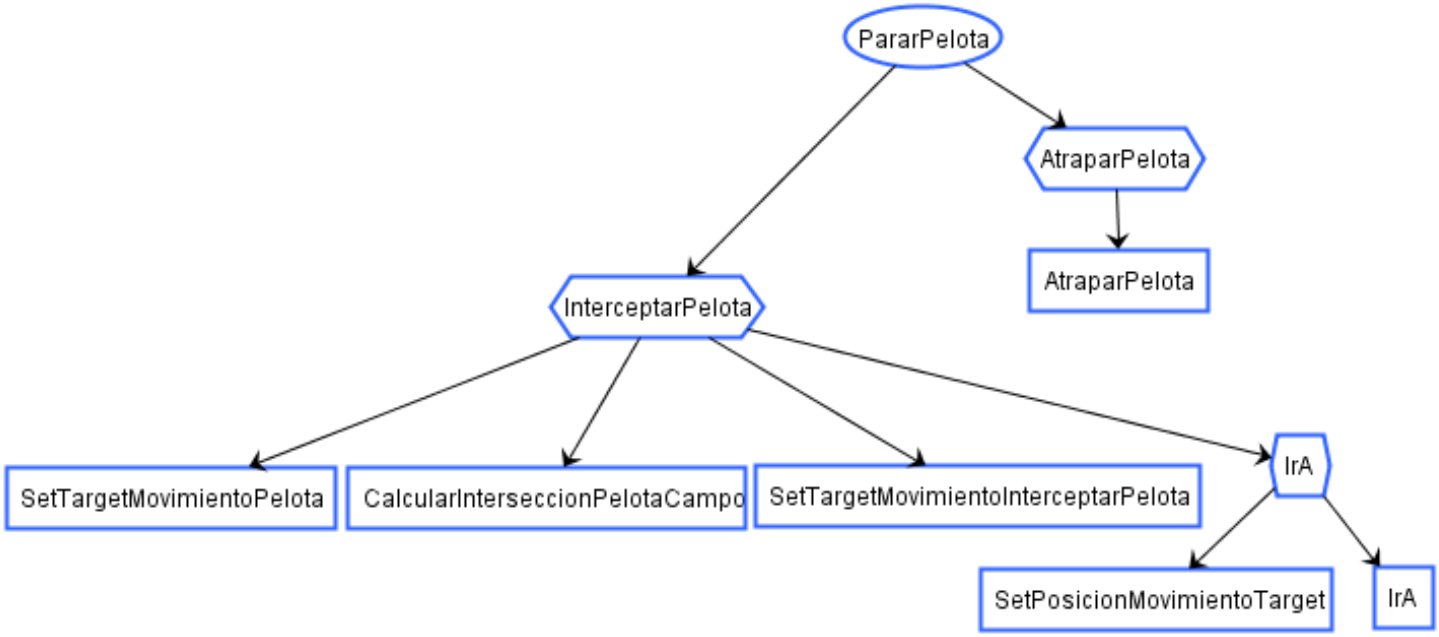


Figura 115: Árbol del comportamiento "Parar la pelota"

El comportamiento de parar la pelota consiste en interceptar la pelota tras un remate a portería por parte del rival.

Para poder interceptar la pelota es necesario anticiparse al movimiento de la misma haciendo una predicción de la posición de la trayectoria de la pelota a la portería más cercana a la posición actual del futbolista. Como esta posición puede variar en función de la velocidad de la pelota y del mismo futbolista es necesario recalcularla en cada iteración.

Si el futbolista logra llegar a dicha posición y la pelota se encuentra lo suficientemente cerca entonces intenta atraparla para hacer suya la posesión.

Proteger la portería

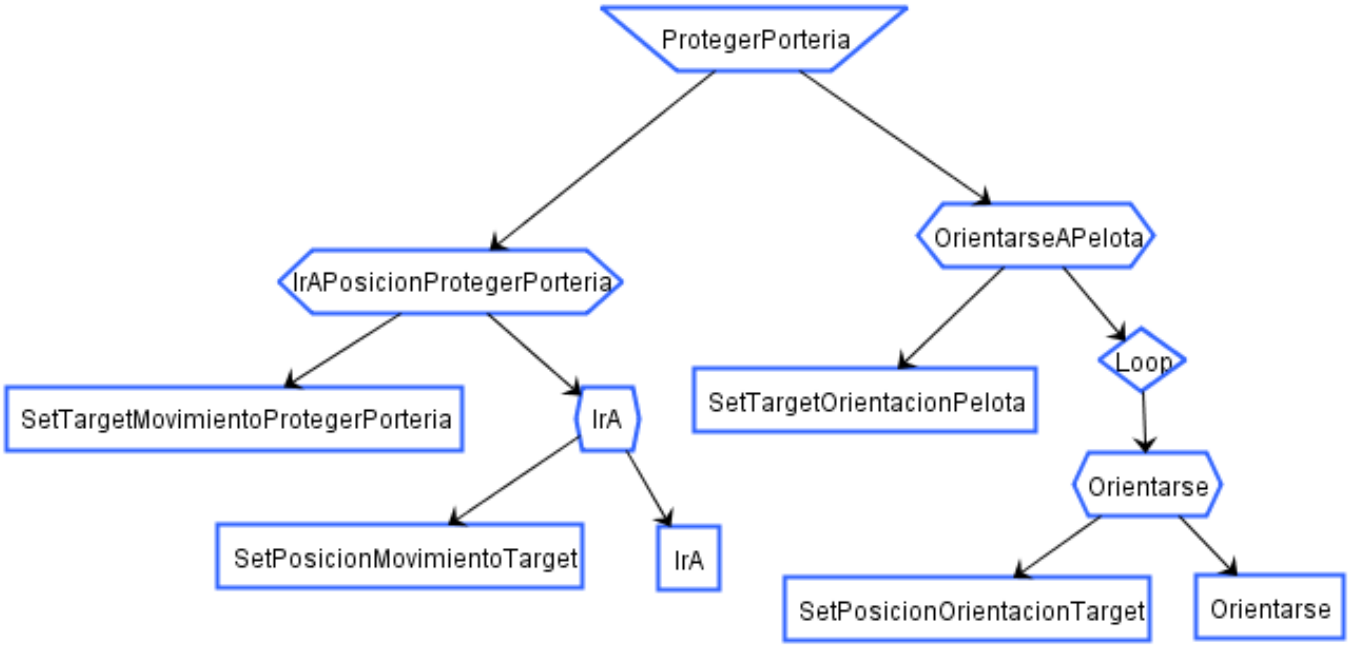


Figura 116: Árbol del comportamiento "Proteger la portería"

La finalidad de este comportamiento es situar al futbolista en una posición que le permita defender la portería.

Su nodo raíz es un comportamiento paralelo que permite ejecutar a la vez los sub-comportamientos “IrAPosicionProtegerPorteria” y “OrientarseAPelota”.

El sub-comportamiento “IrAPosicionProtegerPorteria” tiene como objetivo situar al futbolista en una buena posición para proteger la portería. Para ello primero calcula la posición final donde debe ir el futbolista y luego acerca al futbolista hacia esa posición. Hay que destacar que la posición idónea varía a lo largo del tiempo y necesita ser recalculada en cada iteración, por este motivo no hay un nodo “Loop” que itere indefinidamente el comportamiento “IrA”.

El sub-comportamiento “OrientarseAPelota” tiene como objetivo modificar la orientación del futbolista de manera que mire en todo momento a la pelota que está en movimiento. Para ello primero fija a la pelota como entidad a seguir y posteriormente obtiene su posición para poder orientarse hacia ella.

5.2.4 Tácticas colectivas

Se ha creado un ejemplo para cada uno de los tres tipos de coordinación que soporta el motor del videojuego para demostrar su funcionamiento. En concreto la coordinación necesaria para la realización de los pases se ha realizado mediante una coordinación explicita por pizarra, la táctica colectiva llamada pressing mediante el uso de las tácticas que ofrece el motor y la basculación mediante coordinación implícita.

Pases

Para realizar un pase es necesaria la coordinación del futbolista que posee la pelota con otro futbolista para que cuando se inicie el pase, el jugador receptor esté preparado para recibir la pelota.

Para realizar la coordinación, el futbolista que chuta la pelota marca en el conocimiento compartido el futbolista que ha decidido que debe ser el receptor del pase. De esta manera tanto el futbolista seleccionado como el resto son conocedores de este hecho.

Para evitar que el futbolista que chuta la pelota intente recuperarla por ser el jugador más cercano en ese momento, se marca en el conocimiento compartido como chutador y se crea una rama especial en el árbol de objetivos para seleccionar el objetivo que debe alcanzar un futbolista en este estado.

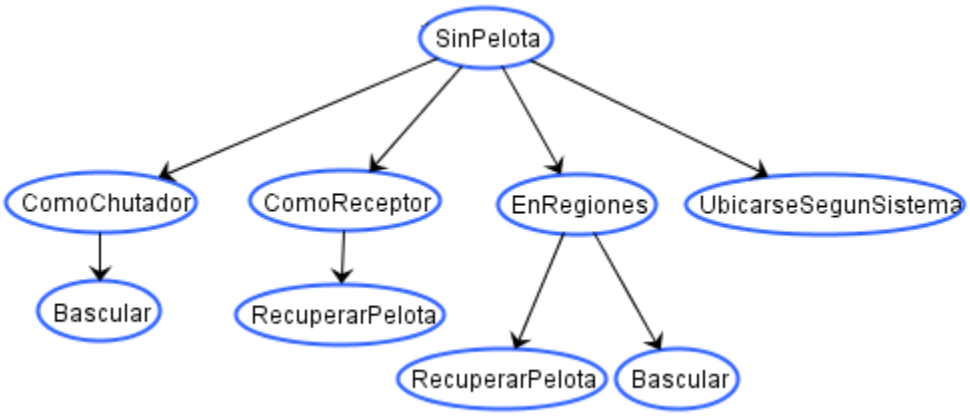


Figura 117: Fragmento del árbol de objetivos para gestionar los pases

Por lo tanto la coordinación se obtiene a partir de la utilización del conocimiento compartido y de la adaptación del árbol de objetivos de los futbolistas para tratar dicha situación.

Pressing

El pressing es una táctica colectiva que consiste en ejercer presión sobre el rival para recuperar la pelota. No es individual ya que al futbolista más cercano al rival con pelota se le suman otros futbolistas de su equipo que están listos para intervenir rápidamente si fracasa en su intento de recuperar la pelota.

Por lo tanto podemos ver que es necesaria la coordinación de varios futbolistas pero no es deseable que muchos futbolistas del mismo equipo se sumen a esta táctica porque entonces dejarían zonas del campo desprotegidas.

Para realizar este tipo de coordinación en que es necesario limitar el número de futbolistas que participan en la misma, se ha creado un objetivo compartido en el árbol de objetivos de los futbolistas que mantiene una referencia a una táctica que representa la táctica de pressing almacenada en el gestor de tácticas.

Para definir una táctica en el motor de inteligencia artificial implementado es necesario definir el número mínimo y máximo de participantes, definir que comportamientos deben ejecutar los participantes, definir una política para seleccionar a los mejores candidatos y por ultimo definir cuando se da por finalizada la táctica.

Táctica	Pressing
Mínimo de participantes	2

Máximo de participantes	3
Comportamientos	Todos los participantes ejecutarán el comportamiento de recuperar la pelota.
Mejor candidato	Un nuevo candidato reemplaza a un participante que esté ejecutando la táctica cuando el nuevo se encuentra a una distancia inferior de la pelota.
Finalización de la táctica	Cuando el equipo de los participantes de la táctica recupera la posesión de la pelota.

Figura 118: Tabla de especificación de la táctica "Pressing"

Para que los futbolistas puedan utilizar la táctica de pressing es necesario crear un nuevo objetivo compartido en el árbol de objetivos que apunte a la táctica para que cuando un futbolista evalúe sus objetivos pueda registrarse como candidato en la táctica si cumple con las precondiciones necesarias.

Como se trata de una táctica defensiva, se ha creado el objetivo compartido pressing como sub-objetivo de defender. Como se puede observar en la siguiente figura, cuando un futbolista evalúa los sub-objetivos del objetivo defender primero evaluará la táctica de pressing y si cumple las precondiciones se marcará como candidato. Si aun cumpliendo las precondiciones no hay candidatos suficientes para ejecutar la táctica, el futbolista pasará a evaluar el sub-objetivo bascular.

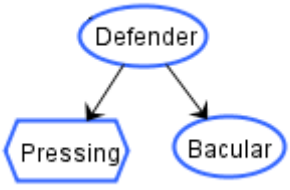


Figura 119: Fragmento del árbol de objetivos de un jugador de campo para gestionar las tácticas

Basculación

La basculación consiste en amoldar la ubicación dependiendo de donde estén la pelota y el resto de futbolistas del equipo. Es un tipo de coordinación implícita ya que no se obtiene mediante la comunicación de información mediante el conocimiento compartido ni es controlada por las tácticas del motor.

Cuando todos los futbolistas basculan correctamente consiguen mantener una distancia adecuada entre ellos para poder cubrir el campo de fútbol manteniendo su formación predeterminada.

Esta coordinación se obtiene a partir del comportamiento de basculación que calcula hacia donde debe ir el jugador respecto a la pelota y luego aplica una fuerza inversamente proporcional a la distancia que separa al futbolista de sus compañeros más cercanos. De esta manera si dos jugadores del mismo equipo se encuentran en posiciones muy cercanas el comportamiento de bascular lo que hará es separarlos pero teniendo en cuenta también al resto de compañeros.

6 Planificación y costes

6.1 Planificación

Al inicio del proyecto se hizo una planificación con todas las tareas a realizar y la duración estimada de cada una de ellas. Las tareas han sido agrupadas en 5 fases:

- **Fase 1: Definición del proyecto.** El objetivo es definir el proyecto así como analizar las soluciones existentes.
- **Fase 2: Desarrollo del juego.** Su objetivo es desarrollar un videojuego simple que permita demostrar el funcionamiento del motor de inteligencia artificial desarrollado.
- **Fase 3: Desarrollo del motor de inteligencia artificial.** El objetivo es desarrollar un motor de inteligencia artificial reutilizable.
- **Fase 4: Desarrollo de la inteligencia artificial específica**. El objetivo es desarrollar la inteligencia artificial necesaria en un videojuego de fútbol utilizando el motor de inteligencia artificial desarrollado.

  Fase 5: Documentación. El objetivo de esta fase es recopilar toda la documentación generada durante el desarrollo del proyecto.

Cada fase del proyecto ha seguido un desarrollo iterativo en el que constantemente se ha ido revisando y replanificando el trabajo según las nuevas necesidades que han ido surgiendo.

A continuación se muestra el listado de tareas agrupadas en fases con la estimación del tiempo dedicado.

	Tarea	**Días**	**Horas**
FASE 1	**Definición del proyecto**	**10**	**80**
	Definición y gestión	4	32
	Planificación	1	8
	Estudio de antecedentes	5	40
FASE 2	**Desarrollo del juego**	**20**	**160**

	Puesta a punto del juego	3	24
	Subsistema de gráficos	4	32
	Núcleo del juego	6	48
	Subsistema de físicas	5	40
	Procesamiento de la entrada	2	16
FASE 3	**Desarrollo del motor de inteligencia artificial**	**25**	**200**
	Agentes inteligentes	4	32
	Toma de decisiones a corto plazo	10	80
	Toma de decisiones a largo plazo	3	24
	Tácticas colectivas	6	48
	Integración del motor	2	16
FASE 4	**Desarrollo de la inteligencia artificial especifica**	**23**	**184**
	Análisis de comportamientos en el fútbol	5	40
	Representación del conocimiento	3	24
	Objetivos	2	16
	Comportamientos individuales	8	64
	Tácticas colectivas	5	40
FASE 5	**Documentación**	**12**	**96**
	Memoria	10	80
	Presentación	2	16
Total		**90**	**720**

Figura 120: Tabla con la planificación de las tareas del proyecto

Aunque las fases están claramente diferenciadas ha habido un solapamiento temporal en el desarrollo de las mismas por la necesidad de poder probar y evaluar cada una de las nuevas funcionalidades del motor.

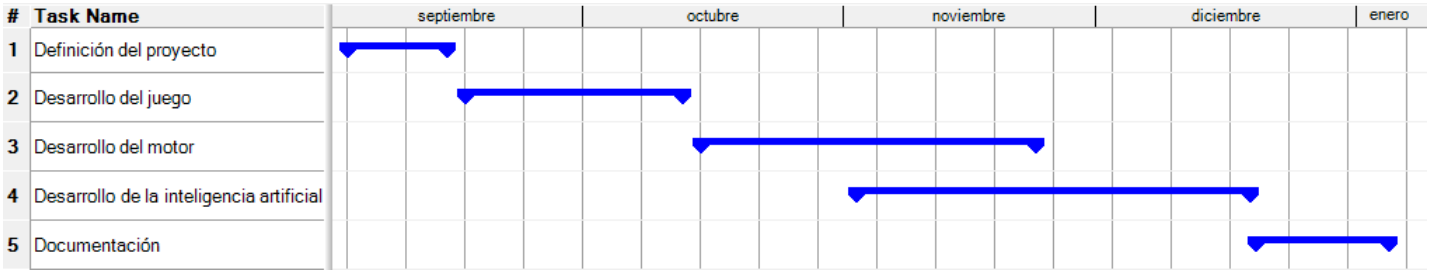


Figura 121: Diagrama de Gantt con la distribución temporal de las fases del proyecto

La planificación de las tareas se ha realizado como si de un proyecto empresarial se tratara. Con este objetivo se han repartido las diferentes tareas entre 4 perfiles profesionales diferentes.

- **Jefe de proyectos.** Encargado de gestionar el proyecto.
- **Analista/Diseñador.** Encargado de analizar el problema y diseñar su solución.
- **Programador.** Encargado de realizar la implementación.
- **Artista.** Encargado de desarrollar el material artístico necesario.

A continuación se muestra una estimación del tiempo dedicado a cada tarea por cada uno de los perfiles que participan en el proyecto:

	Tarea	**J**	**A/D**	**P**	**A**
FASE 1	**Definición del proyecto**	**40**	**40**		
	Definición y gestión	32			
	Planificación	8			
	Estudio de antecedentes		40		
FASE 2	**Desarrollo del juego**		**50**	**102**	**8**
	Puesta a punto del juego			24	
	Subsistema de gráficos		8	16	8
	Núcleo del juego		16	32	
	Subsistema de físicas		24	16	
	Procesamiento de la entrada		2	14	
FASE 3	**Desarrollo del motor de inteligencia artificial**		**73**	**127**	
	Agentes inteligentes		16	16	
	Toma de decisiones a corto plazo		28	52	
	Toma de decisiones a largo plazo		8	16	
	Tácticas colectivas		18	30	
	Integración del motor		3	13	
FASE 4	**Desarrollo de la inteligencia artificial especifica**		**92**	**92**	
	Análisis de comportamientos en el fútbol		40		
	Representación del conocimiento		8	16	
	Objetivos		4	12	

	Comportamientos individuales		24	40	
	Tácticas colectivas		16	24	
FASE 5	**Documentación**	**36**	**60**		
	Memoria	20	60		
	Presentación	16			
Total		**76**	**315**	**321**	**8**

Figura 122: Tabla con la planificación de las tareas del proyecto por perfiles profesionales

6.2 Costes

6.2.1 Coste de personal

El coste de personal incluye el coste del trabajo realizado por los trabajadores que han participado en el proyecto. A partir de la dedicación de cada perfil en el proyecto y su coste a la hora se ha obtenido un coste estimado de 17115€.

Rol	**Horas**	**Precio/hora**	**Coste**
Jefe de proyecto	76	35€/h	2660€
Analista/Diseñador	315	25€/h	7875€
Programador	321	20€/h	6420€
Artista	8	20€/h	160€
Total	**720**		**17115€**

Figura 123: Tabla con los costes de personal

6.2.2 Coste de materiales

El coste de materiales incluye el coste derivado de las licencias del software utilizado en el proyecto así como el hardware utilizado durante el desarrollo.

Concepto		**Coste**
Licencias		**550 €**
	Ogre3D	0 €
	Bullet	0 €
	Blender	0 €
	Visual Studio 2008	550 €
Hardware		**800 €**
	PC desarrollo	800 €
Otros		**200 €**
Total		**1.550 €**

Figura 124: Tabla con los costes de materiales

6.2.3 Coste total

El coste total del proyecto equivale e la suma del coste de personal y el coste de materiales.

Concepto	**Coste**
Personal	17.115 €
Materiales	1.550 €
Total	**18.665 €**

Figura 125: Tabla con el coste total del proyecto

7 Conclusiones

Una vez finalizado el proyecto se han revisado los objetivos y se ha podido comprobar que se han cumplido todos los objetivos que fueron marcados al inicio del proyecto.

- **Desarrollo del juego e integración de los motores gráficos y de físicas.**

  Se ha desarrollado un videojuego de fútbol sencillo donde se ha podido integrar el motor de inteligencia artificial y que permite o bien controlar a un equipo de fútbol o bien enfrentar a dos equipos controlados por la inteligencia artificial.

- **Desarrollo del motor de inteligencia artificial.**

  El motor desarrollado es capaz de gestionar los agentes inteligentes que representan las entidades inteligentes de un videojuego donde las condiciones cambien constantemente y se requiera una coordinación de los mismos.

  En concreto es capaz de gestionar el conocimiento de los agentes así como su toma de decisiones tanto a corto como a largo plazo de una manera sencilla y reutilizable. También ofrece mecanismos para gestionar la coordinación de los agentes ya sea implícita o explícita.

  Por lo tanto el motor desarrollado establece un marco de trabajo que facilita el desarrollo de inteligencia artificial en videojuegos donde se requiera la toma de decisiones en entornos dinámicos y la coordinación entre las entidades inteligentes.

- **Desarrollo de la inteligencia artificial específica para los futbolistas.**

  Se ha desarrollado un conjunto de objetivos, comportamientos y tácticas colectivas que han permitido evaluar el correcto funcionamiento del motor de inteligencia artificial permitiendo a los agentes inteligentes actuar de una manera adecuada a cada situación dentro del contexto de un juego de fútbol.

7.1 Trabajo futuro

Aunque el motor de inteligencia artificial implementado ya establece un marco de trabajo que facilita el desarrollo y reutilización de la inteligencia artificial para un videojuego, una posible mejora sería la creación de un editor externo que permitiera diseñar tanto los árboles de objetivos como los árboles de comportamiento. Este editor permitiría visualizar y diseñar los árboles de una manera más sencilla agilizando así el desarrollo de la inteligencia artificial. La información generada por el editor sería leída por el motor el cual durante la inicialización del videojuego crearía los árboles definidos externamente.

7.2 Valoración personal

El desarrollo de este proyecto me ha permitido aplicar muchos de los conocimientos adquiridos durante la carrera de Ingeniería en Informática. Por ejemplo, la asignatura Aplicaciones de la Inteligencia Artificial (AIA) me sirvió de introducción al mundo de la creación de agentes inteligentes y junto con la asignatura de Videojuegos (VJ) fueron la fuente de inspiración para este proyecto.

Los conocimientos adquiridos en las asignaturas de Ingeniería del Software (ES1 y ES2) también los he podido aplicar durante las fases de análisis y diseño de cada uno de los componentes del videojuego. De igual manera ha sido de gran utilidad la teoría de Visualización e Interacción Gráfica (VIG) para poder configurar correctamente el motor gráfico.

Por lo tanto valoro muy positivamente la realización de este proyecto ya que me ha permitido tanto hacer una síntesis de gran parte del conocimiento adquirido durante la carrera como introducirme en el mundo del desarrollo de inteligencia artificial para videojuegos.

8 Bibliografía

8.1 Inteligencia artificial

- Rabin, Steve. AI Game Programming Wisdom. 2002.
- Schwab, Brian. AI Game Engine Programming. 2004.
- Russell, Stuart y Norvig, Peter. Artificial Intelligence, A Modern Approach. 1995.
- Rollings, Andrew y Morris, Dave. Game Architecture and Design. 2003.
- Buckland, Mat. Programming Game AI by Example. 2004.
- Bourg, David y Seeman Glenn. AI for Game Developers. 2004.
- Pillosu, Ricard. Coordinating Agents with Behaviour Trees.
- Dyckhoff, Max. Evolving HALO's Behaviour Tree AI.
- Dyckhoff, Max. Decision Making and Knowledge Representation in HALO3
- Champandard, Alex y Dawe, Michael y Hernandez-Cerpa, David. Behaviour Trees: Three Ways of Cultivating Game AI.
- Understanding the Behavior Trees. <http://aigamedev.com/open/articles/bt-overview/>
- Autómatas de estados finitos Jerárquicos para el control de comportamientos <http://www.starcostudios.com/blog/2010/02/automatas-de-estados-finitos-jerarquicos-para-el-control-de-comportamientos-hfsms/>
- Hierarchical State Machine. <http://www.eventhelix.com/RealtimeMantra/HierarchicalStateMachine.htm>
- Decision Trees. <http://www.aihorizon.com/essays/generalai/decision_trees.htm>

8.2 Fútbol

- Expert Football.
  <http://expertfootball.com/>
- Gran Fútbol.
  <http://www.granfutbol.com/>
- Tácticas de Fútbol.
  <http://www.tacticasdefutbol.com/>

8.3 Herramientas

- C++. A brief description.
  <http://www.cplusplus.com/info/description/>
- OGRE – Open Source 3D Graphic Engine.
  <http://www.ogre3d.org/>
- Bullet Physics Library.
  <http://www.bulletphysics.org/mediawiki-1.5.8/>
- Blender.Tutorials.
  <http://www.blender.org/education-help/tutorials/>

8.4 Otros

- Wikipedia.
  <http://es.wikipedia.org/>